Democratizing High-Performance DSL Development with the BuildIt Framework

by

Ajay Brahmakshatriya

B.Tech., IIT Hyderabad, 2016
S.M., Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by: Ajay Brahmakshatriya
Department of Electrical Engineering and Computer Science
August 15, 2025

Certified by: Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Democratizing High-Performance DSL Development with the BuildIt Framework

by

Ajay Brahmakshatriya

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2025 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Modern high-performance software from a variety of domains relies on hand-written and hand-optimized libraries to obtain the best performance. Besides general fine-grained operators that can be composed to write entire applications, these libraries also provide coarser-grained fused and hand-optimized operators that are much faster due to being optimized for a specific sequence of operations. However, as application needs keep growing, library writers are not able to keep up and have to make the tradeoff of either sacrificing performance or generality. Domain-specific languages or DSLs are able to break this tradeoff by automatically generating the best implementation for any arbitrary sequence of operations specified by the end user. However, DSL compilers suffer from a bigger challenge that they require a lot of compiler knowledge to implement parsers, IR, analysis and transformations, and code generation, which is outside the scope of a typical domain expert.

To make compiler technology and the benefits of code-generation more accessible to domain experts, I propose the use of multi-stage programming to allow writers to write library-like code while also combining it to generate the most efficient implementation for any whole program. In this thesis, I discuss the design of different multi-stage programming systems, the benefits and drawbacks. Next, I propose Re-Execution Based Multi-Staging (**REMS**) that addresses a critical flaw in many imperative Multi-Staging systems - the side-effect leak problem. I introduce BuildIt, an implementation of **REMS** in one of the most popular languages for writing high-performance applications, C++ in a type-based, lightweight way without changing the compiler. I describe the internals of BuildIt and how it implements the key features of **REMS**.

Furthermore, I describe a set of extensions implemented on top of BuildIt that facilitate the development of high-performance DSLs with ease. I show the application of BuildIt to create three DSLs - EasyGraphit, NetBlocks, and BREeze that target graph analytics, ad-hoc network protocol generation, and Regex matching. All these case studies show 10-100x reduction in the amount of effort required to implement these DSLs that perform on-par with or better than state-of-the-art compiler frameworks while targeting diverse architectures like CPUs and GPUS.

Finally, I introduce D2X, a system that is designed to add extensible and contextual debugging support to DSL implementations without having to make any changes to off-the-shelf debuggers or mess with complex debugging formats. Next, I show how applying D2X to the BuildIt system greatly improves the debugging experience for all DSLs written with BuildIt.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

To mom and dad

Acknowledgments

I would like to thank Saman Amarasinghe, my advisor, mentor, and definitely a friend through the last seven years, for guiding and teaching me almost everything I know about research. Saman has been a constant patient force through this whole journey, both with me and all the problems I have dealt with, research and otherwise. He has guided me inside the walls of his office and MIT and outside. I remember very clearly our discussion in the small cafe in Edinburgh, where you sat with me for two hours and helped me navigate through a very difficult dilemma that had been weighing me down for more than a year. I did walk out of that conversation with a satisfied mind, and I thank you for that. Saman has been a great collaborator in research. His optimism and problem-first approach definitely complements my style of research, not only helping me achieve success but also helping me learn how to be a better researcher. Looking back, I am absolutely certain that joining your group was the best decision for me. I hope to continue collaborating with and learning from Saman in the years to come.

I would also like to thank my committee members, especially Manya Ghobadi, for being an amazing collaborator and mentor for the last 4 years. Your passion and dedication inspire me to be a better researcher every day. I would also like to thank Srini Devadas, my academic advisor and committee member, who has always been so kind and welcoming in my time here at MIT. I always looked to you for when I needed advice, and I hope to be able to reach out to you in the future for the same. I would also like to thank Martin Rinard for the great discussions on the floors of Stata and all his feedback on my thesis and work.

I do want to take this opportunity to thank all my previous collaborators, especially. Changwan Hong, Tom Chen, and Yunming Zhang for not only teaching me so much about compilers and performance but also being very great friends. My time at MIT wouldn't have been so enjoyable without your company. The afternoon and late-night walks with Tom have been moments of so many research epiphanies. I would like to thank all the members of the COMMIT group who have been wonderful to work with and have been great friends. Manya Bansal, whom I have known for only two years but already inspires me with her energy, brilliance, and enthusiasm. I would like to thank all the master's and undergrad students I have had the privilege to mentor - Tamara Mitrovska, Tasmeem Reza, Nicholas Dow, Dhruva Saraff, Kaustubh Dighe, Vedant Paranjape, Aryan Kumar, Katherine Mohr, Alexandra Dima, Claire Hsu, Chris Rinard, and Alice Chen. I have learned a lot more from you than I could teach you, and I hope you have enjoyed working with me. I would also like to thank Mary McDavitt for being so kind and welcoming, and being just an email away for helping with everything I needed.

I would like to thank all my friends at MIT and outside who have made this long journey fun and exciting. Thanks for making this city feel like home. Especially my two best friends, Saurabhchand Bhati and Pratik Bhatu, who have been with me through thick and thin and have been the two pillars

I can always rest my burdens on. This wouldn't have been possible without you two. I would also like to thank my friends at the MIT Graduate Student Union for reminding me that it is our moral obligation to fight for what we believe is right, no matter the odds.

And above all, I would like to thank my family - my mom, Sangita Brahmakshatriya, my dad, Rajendra Brahmakshatriya, and my sister Jai Brahmakshatriya for all your love and sacrifices that have brought me here today. I am extremely grateful for holding me tight when I was scared and things got dark, for cheering me on at every success. Mom, every single day through the seven years, I have wished to be around you to be able to hug you and feel your warm embrace. Thank you for loving me and being patient with me despite everything. Dad, I want to thank you for everything you have taught me, from using all the tools to making tough life choices, and above all, staying true to one's principles. Jai, thanks a lot for being so caring all my life and always being supportive of all my choices. I am truly indebted to all of you.

This research spanning a duration of seven years has been supported by several grants. DARPA SDH Award HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, NSF under award CCF-2107244, Intel/NSF award CCF-2217064, DARPA award HR0011-20-9-0017, ACE Center for Evolvable Computing Research Center, a JUMP 2.0 Center co-sponsored by SRC, DARPA and other companies, DARPA PROWESS Award HR0011-23-C-0101, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, Sloan fellowship FG-2022-18504. ,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Every problem in Computer Science can be solved with a level of indirection and every problem in Software Engineering can be solved with a step of code-generation

1.1 Overview

The world of software engineering and programming in general is extremely heterogeneous both in terms of goals or priorities and in the kind of techniques used to make progress towards those goals. Even so, no matter the area or the level of expertise, almost everybody can agree that faster software is better. It is not a surprise, then, that modern software is incredibly fast. This growth in performance cannot be attributed to just the increase in the number of transistors that can be packed in a unit area, higher clock speeds, or the newer and fancier architectural innovations with hardware accelerators tailored to high-performance application needs, or the tremendous amount of effort being invested into building better and faster software technologies or even just the advancement in AI and AI-powered programming techniques but the combination of all these together. Innovations in hardware have opened up new optimization possibilities in software libraries and programming languages, which have brought AI to a critical threshold where it is not just learning from the software that is out there but also contributing back to the ecosystem and making it, in turn, more optimized and creating a positive feedback loop. Software has ushered into an era where computations previously thought impossible or unfeasible to be completed in a reasonable amount of time have become possible. In 2019, the team working on the Event Horizon Telescope managed to process about 5 petabytes of data to construct the very first image of a black hole [39]. Although this took a whole 2 years after the very first observations had started, such a large amount of astronomical data had never been processed before. Not too long since then, we have large language models like Deepseek-V3 with about 671 billion parameters being pre-trained on about 14.8 trillion tokens in about 55 days [173]. Other popular models like Gemini2 [174], ChatGPT5 [134], LLaMA 3.1 [127] are consuming a similar magnitude of data and are being trained on 1000s of GPUs over days and months. Such large-scale computations are becoming common across domains like bioinformatics, climate modeling, cryptography, databases, and graphics, among others. Outside of data-centers,

even consumer electronics and devices have seen a sharp rise in compute capabilities, with mobile devices equipped with modern neural processing units that can run entire neural network models like ResNet50 [84] locally and even process up to 50 images per second [97]. As new domains open up, the demand for performance is ever-increasing, and this demand falls on the shoulders of developers who are experts in these domains and general software engineers alike.
It is a general understanding in the High-Performance Computing (HPC) community that as the need for performance grows, the amount of effort required on the part of the developer to obtain the performance grows rather quickly. As the general low-hanging fruits in the optimization space are plucked, developers have to look for specialized optimizations for the specific problems. These range from techniques to improve the complexity with better algorithms and data structures, exploiting massively parallel hardware like multi-threaded CPUs, GPUs, TPUs, and other modern accelerators, to performing low-level constant factor improvements by optimizing for memory accesses, instruction throughput, vectorization, and so on. It is safe to say that when it comes to high-performance software development, the effort of implementation required is directly proportional to the expected performance. This increase in effort is often super-linear since the optimization techniques that take you from 90% performance to 99% are minuscule in terms of complexity and effort as compared to the techniques required to go from 99% performance to 99.9% performance.

1.1.1 Libraries as Means to Reduce Effective Effort

Developers everywhere struggle with similar challenges, trying to get the best performance while minimizing the development time and effort. Not very surprisingly, often different developers working on different problems have to deal with the exact same sub-problem. For example, an operation like matrix multiplication has applications in machine learning, graphics and image processing, finite-element simulations, robotics path finding, among others. Total efforts can thus be saved by sharing code bases and optimizations, which leads to reusable libraries and programming abstractions. With this in mind, a better metric to look at is the effective or amortized effort in developing high-performance software. When working on a problem (P) in isolation, the total effort required for optimizing the problem (T_P) is equal to the effort invested by the sole developers of the project (D_P). Since the code-base and the effort are specific to that problem, the effective work (E_P) is simply the sole development effort divided by 1 ($D_P/1$). Thus, for the isolated problem-solving case -

$$E_P = \frac{D_P}{1} = T_P \tag{1.1}$$

However, if the developer of the problem P decides to use a library L_1, parts of the implementation of P can be done by L_1 and the development effort saved would be S_{PL_1}. If the development effort for the library is D_{L_1} and the library is used by N_{L_1} different problems, the new effective effort E_{PL_1} for the problem P while using L_1 is -

$$E_{PL_1} = \frac{D_P - S_{PL_1}}{1} + \frac{D_{L_1}}{N_{L_1}} \tag{1.2}$$

Since the implementation now requires the use of the library, its development cost should also be considered under the effective effort. However, since the library is used by multiple problems, the cost of the library is amortized. For the most popular libraries, the number of users N_{L_1} is a large enough number that, for most practical purposes $\frac{D_L}{N_{L_1}} \to 0$ and $E_{PL_1} \approx \frac{D_P - S_{PL_1}}{1}$. Since the savings are always positive (if not, the developer can simply choose not to use the library), we get $E_{PL_1} < E_P$. Thus, we make the observation that the use of popular libraries reduces the effective effort required to optimize the problem. These efforts can be further reduced by using multiple libraries $L_2, L_3, \ldots, L_K$ such that

$$E'_P = \frac{D_P - S_{PL_1} - S_{PL_2} - ...S_{PL_K}}{1} + \frac{D_{L_1}}{N_{L_1}} + \frac{D_{L_2}}{N_{L_2}} + \ldots \frac{D_{L_K}}{N_{L_K}} \tag{1.3}$$

This also follows intuitively since not every problem should be writing their own memory allocators, their own libc, or their own operating system APIs. Use of these programming abstractions together significantly reduces the effort required by individual programmers. Code sharing in this way also has a secondary effort, where as more projects use a library, the total number of users for the library increases, further reducing its effective cost component for all the projects that use it. As a result, high-performance libraries have not only become popular in a wide variety of domains, but most of the modern high-performance software infrastructure relies on hand-optimized and hand-written libraries.

Table 1.1 shows a non-exhaustive list of libraries that power various domains. Some of these libraries are extremely popular and not only have millions of users but also have a significant investment in terms of man-power and resources from large industries. Pytorch from Meta, TensorFlow from Google, and cuBLAS from NVIDIA are just a few examples of such popular libraries. Furthermore, when targeting different architectures, entirely separate libraries need to be used, or the same libraries have separate implementations optimized for different hardware. cuBLAS, for instance, is specifically optimized for NVIDIA CUDA GPUs, while MPI is specifically built to target distributed systems. Libraries like PyTorch and TensorFlow have support for both CPUs and GPUs, but internally have separately written and optimized implementations.

Since efforts are reduced when parts of the problem are offloaded to libraries, it might be tempting to try to offload most or all parts of the project to libraries. However, for every component moved from a single problem to a library, the savings for the problem are much, much smaller than the effort required to expand the scope of the library. This is mainly due to the combinatorial blowup of operations the libraries have to handle when adding any new operators. I will describe this combinatorial blowup below.

1.1.2 Combinatorial Blowup From Operator Fusion

In this section, I will explain how, as the scope of a particular popular library increases, i.e., as it adds more abstractions to support a wider range of applications, the complexity goes up at an increasing rate. To better understand this, let us start by understanding a feature critical to many high-performance libraries - operator fusion. Most libraries are a collection of data structures and functions that the library exposes to the user. Users write bigger programs by composing together these operations and passing around the data structures between them. However, this composition often comes at a performance cost.

Computation Domain	Libraries
Machine Learning and Deep Learning	TensorFlow, PyTorch, XGBoost, LightGBM, SciKit-Learn, cuML
Numerical and Scientific Computing	NumPy, SciPy, BLAS and LAPACK, Eigen, FFTW, OpenBLAS, BLAS, cuBLAS
Parallel Computing and Distributed Systems	MPU, OpenMP, Dask, Apache Spark, Ray
Graphics and Game Development	Vulkan, OpenGL, Godot, Blender, Assimp
Database and Storage Systems	PostgreSQL, ClickHouse, Redis, Cassandra, LevelDB, RocksDB
Networking and Systems	ZeroMQ, gRPC, libuv, DPDK, QUIC
Cryptography and Security	OpenSSL, libsodium, BoringSSL, WolfSSL
Concurrency and Multi-Threading	TBB, Boost, Asio, libdispatch, CILK
Data Processing and Compression	Apache Arrow, Parquet, Snapp, Zstandard
Bioinformatics	Bioconductor, HTSlib

Table 1.1: List of several scientific computation domains and the hand-optimized libraries that power the applications.

Let us look at the example of a sparse matrix computation library like Eigen [81] or SuiteSparse [54]. These libraries support functions like matrix vector product, matrix matrix product, or matrix matrix dot product, and so on. These libraries also support these computations on a mix of sparse and dense data structures. Suppose our problem needs to compute $A \odot (B \times C)$ where B, C are dense matrices and A is a sparse matrix (this operation is also called Sampled Dense Matrix Matrix Multiplication or SDDMM). One very simple way of implementing this program is by calling a function from any of these libraries that performs dense matrix dense matrix multiplication to produce the intermediate $B \times C$. This intermediate result can then be passed to a sparse matrix dense matrix element-wise product function to compute $A \odot (B \times C)$. While this produces the correct result, let's see what happens during the second operation. Since this step performs an element-wise product, the coordinates where A is zero, the output is also zero, and the result of the intermediate is not used. Since A is sparse, a large fraction of the values are zero, meaning a lot of the computations from the dense matrix-dense matrix product are wasted.

Alternatively, if we implement this operation in a single step such that we iterate through the non-zeros in A and only compute the results of $B \times C$ at those coordinates, we can get much better performance. This is exactly what most high-performance BLAS libraries do. These libraries provide a function called `sddmm` which accepts A, B and C and directly returns the result through a single hand-optimized step. Thus, we observe that performance can be improved by combining two or more operators into a single operator. Such an optimization technique involving combining multiple operators to provide a single coarser-grain operator is called operator fusion. Such operator fusion opportunities are not unique to the sparse domain and exist in many libraries. Even if the matrix A is not sparse, performance can still be improved by iterating through all three matrices at once, saving iteration costs, improving cache performance, and the cost to allocate intermediate results. Thus, to obtain the best performance, library writers often have to keep looking for such opportunities to fuse the operations they support.

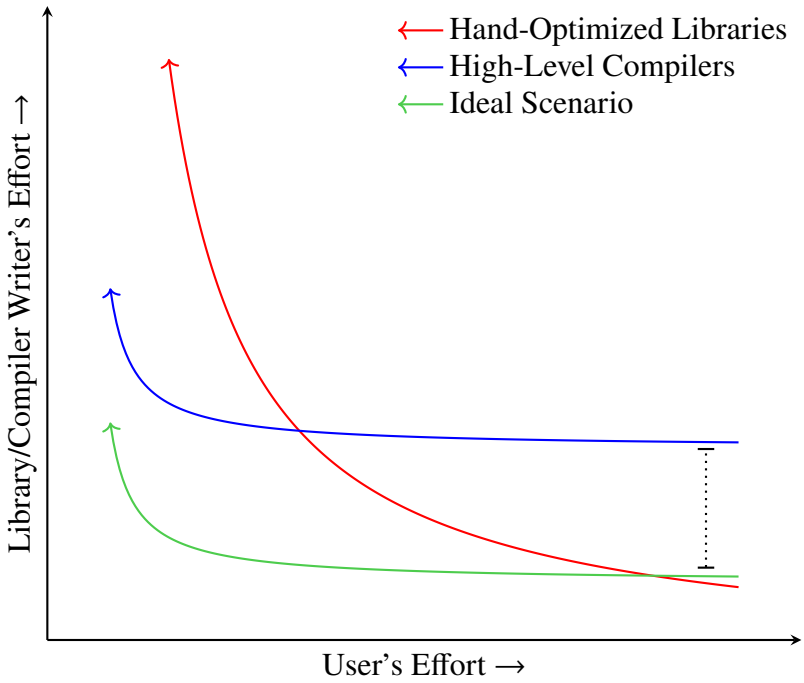


Figure 1.1: The tradeoff between the amount of effort needed from library/compiler developers and the end-users. The dotted line in black shows the constant overhead of getting a compiler abstraction started.

Now, suppose an operation is currently not supported by a library but is required by an application. If the application hand-optimizes that operation for the use case, the developer only has to implement fused functions with the operations they use. However, if the operation was added to the library, the library developer has to produce a fused version with all the other functions they provide, even if they are not used by the application. Thus, adding a new operator implementation to a single application is much cheaper than adding it to an already feature-rich library when trying to obtain the maximum performance.

The red curve in Figure 1.1 shows how the library writer's effort would increase as the effort required by the end-user is reduced, keeping the performance at the best possible performance for a particular application. As we can see, the curve isn't a straight line but follows a super-linear curve. The observation here is that while libraries are a great way to reduce the effort of development and optimization for the end user, supporting the needs of a large number of users adds a significant burden to the library developer due to the combinatorial benefits. Moving operations to libraries is still beneficial up to some point due to the benefits of amortization. However, as the tail end of users is supported, the effective cost increases. I would like the reader to note that this is a project of what would happen if more and more computations are offloaded to libraries. We will see next why the left extreme of the curve isn't practical for most libraries.

1.1.3 Handling Combinatorial Blowup through Code Generation

An alternative approach to handling this combinatorial blowup is through the use of Higher-Level Abstractions that generate code through compiler techniques. These compilers can either be specific

to a certain set of problems, i.e., they can be Domain Specific Languages (DSLs), or can be general enough to combine operators in a way that preserves semantics while improving performance. The main advantage that these compiler techniques offer lies in the fact that the effort required to add a new operator does not scale with the number of operators already present in the language. Instead of providing an exhaustive set of operators fused together, compiler techniques take a different approach where the end-user provides as input the sequence of operations they want to perform to the compiler, and the compiler generates an implementation specialized for those sequences of operations. Thus, as long as the compiler is written in a fairly general way, adding support for a new operator simply boils down to describing to the compiler the semantics of the operator, and it can automatically combine it with all the existing operations and their combinations. This cost grows fairly slowly as more and more operators are moved from being implemented by individual developers to the DSL, as shown by the blue curve in Figure 1.1. This approach is taken by many DSL compilers like TACO [105], Halide [146], GraphIt [201, 25, 23, 198], pytorch [138], Tensorflow [175] and general compiler systems like Tiramisu [7], AlphaZ [196] among others. The TACO DSL is targeted at generating programs in the sparse tensor domain. TACO can take as input an Einsum representing $A \odot (B \times C)$ and generate the most efficient code for the `SDDMM` operation. These DSL compilers can also target different architectures like CPUs, GPUs, and tensor cores, further eliminating the effort required to implement multiple libraries.

However, as seen in Figure 1.1, the compiler approaches have a big problem, which is that implementing them takes a lot of upfront cost. As compared to libraries, compilers are complex systems that have various components like parsers, various intermediate representations, analyses, and transformation passes, and code-generation for various architectures, all of which need to be implemented before even the code for a single operator can be generated for the end-users. These components are not only complex but are also very large. For example, the TACO compiler in its entirety is about 72,000 lines of C++ code, while the Tiramisu compiler is about 349,000 lines of C++ code. Libraries, on the other hand, are fairly simple and can be as small as a few 100 lines of code before the end-users can start using them. Thus, even though the growth in complexity and effort for compiler approaches is slow, the upfront cost is typically a large barrier to overcome for most developers. This is evident from the fact that there are so few compiler approaches to high-performance abstractions as compared to libraries. This problem is especially challenging for domain experts, who are now forced to build compilers despite this being outside their expertise. Meanwhile, compiler experts cannot provide assistance because they are already spread thin between multiple domains.

The consequence of this is that since domain-experts cannot afford to invest the colossal effort required to support all users while maintaining the best performance (left extreme of the red curve in Figure 1.1), the library writers are left with 3 options. The first is to either compromise on generality and limit the scope of their libraries to a few operators, or second, to provide a library with all the operations but require the users to compose them when required and sacrifice performance, or third, invest the upfront cost in building a compiler, acquiring compiler expertise as they go and only after that build something usable. Figure 1.2 shows these three options and their tradeoffs. A general library with basic composable operations is general and supports a lot of applications with minimal developer effort, but sacrifices high performance. A hand-specialized library that carefully picks a smaller subset of operators to hand-specialize and write fused versions of, gets the highest performance again with a reasonable amount of effort, but is not able to support all applications. Finally, a compiler-based approach is able to generate high-performance code for a wide range of

applications but comes at a heavy implementation effort cost.

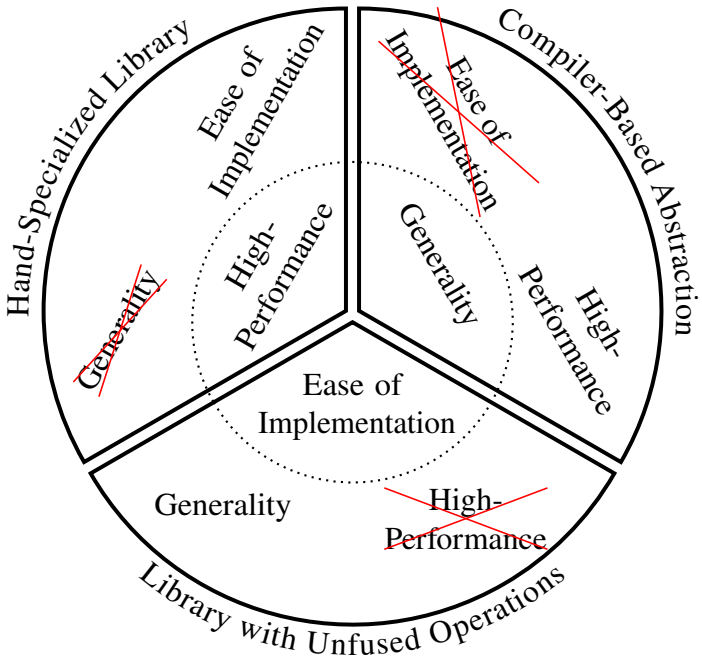


Figure 1.2: The tradeoffs offered by different ways of developing high-performance programming abstractions - a general library with basic composable operators, a hand-specialized library with fused operations, a compiler technique that generates code for an arbitrary sequence of operators. The dotted circle in the center shows the best of all three worlds obtainable using multi-staging.

Ideally, we would want something in the middle, as shown by the dotted circle, that is general, easy to implement, and still has the best performance. The same ideal option is shown using the green curve in Figure 1.1, where the initial effort for the technique is slightly more, if not exactly the same, as a library with simple unfused operators, but also doesn't explode in terms of effort with every new operator added, just like a compiler. The key observation here is that the initial effort needs to be close to a library - an implementation that looks like something that runs and not something that generates code. But the effort shouldn't compound when fusing operators and thus should use code-generation like a compiler. Multi-Staging or Metaprogramming has this property where specialized code can be generated by writing library-like code and specializing it. I will now introduce the idea of Multi-Staging and how it ties in with compiler implementation.

1.2 Multi-Staging as an Alternative to Compiler Development

Multi-Staging or the idea of writing code in multiple stages is a technique known in literature for a long time, first thought to have been introduced and formalized by Walid Taha and Tim Sheard [169]. It can also be thought of as the idea of a program writing or manipulating another program. Hence Multi-Staging is sometimes used interchangeably with the term Metaprogramming. Figure 1.3 a) shows a block diagram for a typical single-stage programming workflow where the programmer

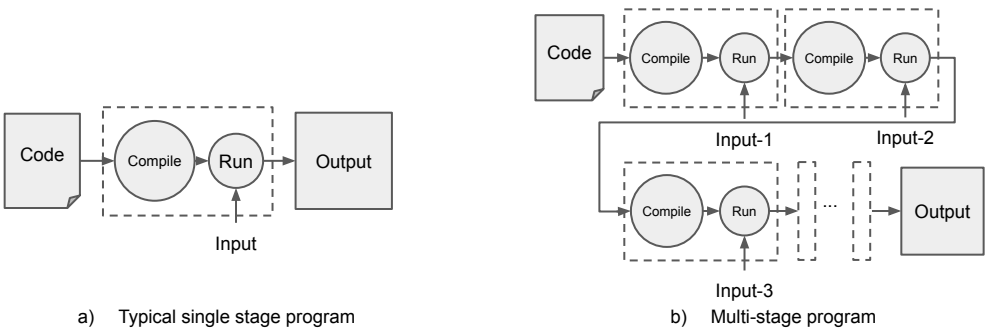


Figure 1.3: Block diagram for a) typical single-stage programming vs b) a multi-stage program with an arbitrary number of stages.

starts by writing some source code. The code is executed after possibly going through a compilation step, depending on the language and the runtime. When this program is executed with the inputs, it generates the final output. Figure 1.3 b) shows a block diagram for a general multi-stage program where the initial source code, when compiled and run with some inputs, produces an output that is another program. This program is then further compiled and executed. This can be repeated multiple times before getting the final output. Chapter 2 will define multi-staging formally, but the high-level idea is that there is at least one step where code is generated as an output from a sufficiently programmable step. This programmable step-generating code is what allows us to get around the combinatorial explosion problem in high-performance libraries. Since the programmable step is what does the combining and fusion of the sequence of the operators that would typically have to be done by the library developer manually, investing a lot of time and effort. While multi-staging can be applied repeatedly over an arbitrary number of stages, for most practical applications related to high-performance abstractions, two stages are enough. The rest of this thesis will keep the primary focus on two stages of multi-staging. I will discuss extensions to support more than two stages.

It is worth noticing that the execution of this step of programmatically combining operators and generating code can be fairly expensive. But multi-staging is typically used when the generation step is run very little often as compared to the generated program, thus amortizing the cost of the generation. This is exactly the case with the high-performance applications. Since the fused operators have to be generated only once when the particular sequence of the operators is decided, but is run hundreds or thousands of times on the actual data.

Multi-Staging can be implemented in a variety of different ways, including different choices of programming languages, different ways of annotating what stage the code should run in, and the amount of changes required to convert a single-stage program into an equivalent multi-stage program, each with its own benefits and drawbacks. These choice of features factors heavily into how well a particular multi-staging framework is able to provide the best of both worlds in terms of implementation effort and being able to support the needs of a variety of end-users while maintaining high-performance. I will discuss the tradeoff of these different kinds of multi-staging in the next Chapter. Moreover, when interleaving code from different stages, especially in an imperative setting, care must be taken to ensure that side effects on the first stage variables are handled appropriately. Many frameworks that do not handle these effects correctly generate incorrect code,

which can hamper the usefulness of the whole system. I will introduce a notion of Re-Execution based multi-staging (**REMS**) to be able to handle these side-effects in a semantically correct way. Furthermore, just being able to generate code programmatically is not enough to obtain the best performance. Multi-Stage systems have to provide the ability to simplify, specialize, analyze, fuse, parallelize, and even make it possible to move around data and other optimizations in an ergonomic way to generate the most optimized code. Finally, the overall user experience of both the developer and the end-user depends not just on the effort and performance but also on the tool-chain support around the language itself, the most important being support for debugging.

To address these challenges in a way that allows transforming existing code bases into high-performance code-generating compilers, I will introduce BuildIt, an implementation of re-execution-based multi-staging in C++, a language most suitable for working with existing high-performance code bases. I will explain the design and implementation of the system in a lightweight way, showing how to add multi-staging to C++ in a seamless type-based way without requiring any changes to the compiler or runtime. Furthermore, I will introduce layers and extensions built on top of BuildIt itself to be able to perform analysis, data-layout optimizations, parallelization for different architectures, and support a wide variety of high-performance domains. Finally, I will show the application of BuildIt to implementing a series of real-world high-performance domain-specific languages that obtain performance similar to state-of-the-art compiler techniques with a 10 to 100 times reduction in code size. I will also discuss the construction of a system that adds rich debugging support to existing and new DSLs with ease.

1.3 Contribution and Scope

In this thesis, I will discuss the differences between different styles of multi-staging relevant to high-performance code generation, their tradeoffs, and the need for re-execution-based multi-staging to generate correct code in a seamless way. I have applied this idea to build a multi-staging system in C++ called BuildIt, which can convert existing high-performance library implementations into compilers with minimal compiler knowledge, thus democratizing the DSL development process. I have built layers on top of BuildIt that further improve the usability and applicability of the system and applied it to create DSLs in a wide variety of domains, including graph processing applications, ad-hoc network stack design, and optimizing regular expression matching, among others. I have also created a companion system D2X which makes debugging of DSLs easier. My specific contributions are:

- A taxonomy of different styles of multi-staging and highlighting the pros and cons they offer, and identifying a common issue that limits the usability of most imperative multi-staging systems for real-world applications - **the side-effects leak problem**.
- Introduce the idea of **Re-Execution Based Multi-Staging (REMS)** and prescribe a procedure to implement **REMS** in a language and technique-agnostic way.
- Construction of **BuildIt, an implementation of REMS in C++**, the most popular language for high-performance in a type-based, lightweight manner. Not only does BuildIt solve the aforementioned issue, but it also presents an extremely seamless API to the developer.

- Extensions and layers on top of BuildIt that are focused on improving the performance of libraries and the code generated from DSLs by enabling **simplification, specialization, analysis, parallelization, staging of data, and hoisting conditions** all while targeting a variety of architectures including demonstrating a methodology for **performing data-flow analysis purely through staging** and the novel **Layout Optimization Layer**.

- Implementation of **three high-performance domain-specific languages** using BuildIt for graph analytics, ad-hoc network protocol generation, and regular expression matching, that not only improve or match the performance of state-of-the-art frameworks but also do so with 10-100x less code size, minimizing the developer effort.

- Implementing D2X, an extensible and contextual debugger for modern DSLs. D2X also works in tandem with BuildIt to provide free debugging support for all DSLs written using BuildIt, without requiring a single line of code to be changed.

In my understanding BuildIt is the first and the most practical framework for rapidly prototyping high-performance domain-specific languages and bringing the benefits of code generation to domains that couldn't have access to it before due to the upfront skill and effort barrier. Domain experts from all areas that have any experience with writing libraries for their domain can now easily convert them into domain-specific languages without having to understand compiler technology like parsers, intermediate representations, explicit analyses and transformations, or code-generation for different architectures. Thus, BuildIt and this thesis make a huge contribution towards democratizing the high-performance DSL development space. However, there are specific areas that still need more work, and some of the limitations need to be addressed -

- One of the most important caveats of developing DSLs this way is that the **developer still has to think about optimizations** that make sense in their domain and implement them. Future work would need to focus on developing a library of reusable and composable optimizations that can be applied to various domains, all implemented using BuildIt. A new DSL developer can then just pick and choose the features/optimizations they need for their domains, further simplifying the DSL development process. One instance of this is the Layout Optimization Layer and the CUDA Kernel Fusion methodology.

- The other downside of developing DSLs this way is that DSLs implemented with BuildIt are embedded in C++ and **thus are limited by the syntax C++ offers**. If languages need specific syntax for their domain, that cannot be implemented by operator overloading and function calls, the developers would find themselves limited. However, these developers can still use the BuildIt system for lowering and optimizations where they would implement their own parser for the new language and then "interpret" the IR using BuildIt to generate the code. More work needs to be done to explore DSLs written this way and the benefits they offer.

- BuildIt is also an ideal fit for languages where most of the **optimizations are implemented as lowering as opposed to lifting**. BuildIt does support lifting small patterns by using *static* variables for looking for these patterns, but for more complicated patterns, the developers would have to extend BuildIt by writing a traditional pass for analyzing and transforming the AST, an approach that is probably limited for use by compiler-experts. BuildIt also has support for basic pattern matching through an expression matching language built in, but

more work needs to be done to support matching and replacement, complex control flow, and statements easily - features that are typical of most high-performance code bases.

- The main focus of BuildIt is on code generation and high-performance optimizations for DSLs. However, DSLs that require some unique type-system or are focused not on performance but on other aspects like visualization, or managing resources or build systems, among others, might find BuildIt not so applicable. More work needs to be done to explore if any of the techniques developed as part of BuildIt might be applicable to other non-performance focused domains.

However, nothing in BuildIt or this thesis is fundamentally opposed to or incompatible with the topics presented above. In fact, BuildIt provides the initial groundwork for answering these questions. **My hope is that this thesis starts the discussion about making compiler technology accessible to non-compiler experts across all domains.**

1.4 Organization of the Thesis

The rest of the thesis is organized as follows:

1. **Chapter 2 - (REMS: Re-Execution Based Multi-Staging)** starts with a taxonomy of different styles of multi-staging and compares their benefits and drawbacks. The chapter also presents a problem that plagues most, if not all multi-staging implementations and proposes a language-agnostic solution through the use of **REMS**

2. **Chapter 3 - (BuildIt: Re-Execution based Multi-Staging in C++)** introduces BuildIt, an implementation of **REMS** in C++, and explains why C++ is an ideal choice for targeting high-performance domains through an approach like **REMS** and multi-staging. This chapter also serves as a language manual for anyone looking to use, extend, or build on top of BuildIt.

3. **Chapter 4 - (Implementing REMS in BuildIt)** covers the challenges of implementing in C++ a compiled language with little to no support for introspection. This chapter walks step by step through how each component of BuildIt is implemented and how everything comes together to provide the user with an API that is seamless but powerful.

4. **Chapter 5 - (BuildIt Extensions for High-Performance Libraries and DSLs)** talks about what features are required for actually implementing high-performance DSLs using multi-staging and demonstrates how these can be implemented either as an extension to BuildIt or simply on top of the API BuildIt already provides. The chapter discusses these by applying these techniques to a mini array DSL called BArray.

5. **Chapter 6 - (Case Studies for Developing DSLs with BuildIt)** discusses the motivation, design, and implementation of three unique domain-specific languages on top of BuildIt for targeting the domains of graph analytics, ad-hoc network protocol generation, and regular expression matching. This chapter also evaluates the performance and the effort required to implement these DSLs, comparing them to the effort required to implement similar traditional compilers and libraries wherever available.

6. **Chapter 7 - (Extensible and Contextual Debugging for DSLs)** discusses the need for also focusing on improving debugging support for DSLs alongside optimizations and code generation, and discusses the implementation of D2X, a debugger focused mainly on DSLs that need to describe custom information specific to the DSL. This chapter also discusses the experience of applying the framework to a traditional DSL compiler and the BuildIt framework itself.

7. **Chapter 8 - Related Works** discusses the related work in all the areas of DSL development, multi-staging, high-performance compiler optimizations, GPU and parallel CPU code generation, graph analytics, networking, regular expressions, and debugging.

8. **Chapter 9 - Conclusion** concludes this thesis and talks about the potential future directions of research for both BuildIt and DSL development itself.

Chapter 2

REMS: Re-Execution Based Multi-Staging

In the previous chapter, we saw that multi-stage programming or multi-staging is a great way to balance the benefits of hand-optimized libraries and compiler-based techniques by providing a good coverage for applications without sacrificing performance, without requiring colossal amounts of effort to implement full-fledged compilers. In this chapter, I will dive deeper into multi-staging, starting with its definition, followed by a taxonomy based on various design choices, their benefits and drawbacks, and motivate the need for a new way of doing multi-staging based on Re-Execution. Let's start by looking closely at what multi-staging really means. In programming, multi-staging is the general idea where programs produce other programs when evaluated. The idea has similarity with the notion of metaprogramming, partial evaluation, or simply code generation. Multi-Staging is sometimes also used to describe programs that manipulate other programs. A compiler or a transpiler is an example of a multi-staging system where the execution of the compiler produces an output program that can be executed or compiled again.

Programs can be made to output programs in many ways. One way to do this is by evaluating the program for a subset of inputs it accepts. This is also called partial application or partial evaluation [169, 151]. Partial application allows us to specialize a program for specific inputs while still preserving the same interface and implementation if done correctly. For the entirety of this thesis, we will be looking at this form of multi-staging, its benefits, implementation, and applications.

2.1 Formalizing Multi-Staging

Let us define partial application multi-staging formally. Consider a program $\mathbb{P}$ that accepts parameters $\{x_0, x_1, x_2, \ldots, x_n\}$ where each x_i is of type T_i and produces an output y of type T_y. For any full set of arguments $A = \{a_0, a_1, a_2, \ldots, a_n\}$, such that each a_i is of type T_i when $\mathbb{P}_S$ is evaluated with A, it produces an output b_A with type T_y. That is,

$$\mathbb{P} : T_0 \times T_1 \times \cdots \times T_n \rightarrow T_y$$
$$\forall A = a_0, a_1, a_2, \ldots, a_n, \text{ with } a_i \in T_i, \text{ we have: } eval(\mathbb{P}, A) \rightarrow b_A$$

With 2-stage multi-staging, for every subset S of $\{0, 1, 2, ...n\}$, there exists an equivalent program $\mathbb{P}_S$ which only accepts the parameters $\{x_i : i \in S\}$. For all arguments A, when $\mathbb{P}_S$ is evaluated with arguments $\{a_i : i \in S\}$, it produces an output program $\mathbb{Q}_{SA}$ which accepts parameters $\{x_i : i \in D\}$ and produces the output y, where D is defined as the complement of S, $D = \{0, 1, 2, ...n\} - S$. This program $\mathbb{Q}_{SA}$ when evaluated with the rest of the inputs $\{a_i : i \in D\}$ returns the same output b_A as $\mathbb{P}$. The evaluation procedure of $\mathbb{P}_S$ and $\mathbb{Q}_{SA}$ can be different from each other and from the evaluation of $\mathbb{P}$. Hence, we will use two different evaluation functions $eval_{static}$ and $eval_{dynamic}$.

$\forall S \in \{0, 1, ...n\}\ \exists\ \mathbb{P}_S : \{T_i : i \in S\} \rightarrow \{T_i : i \in D\} \rightarrow T_y$ where $D = \{0, 1, 2, ...n\} - S$
$\forall A = \{a_0, a_1, a_2, ...n\}$ we define $A_S = \{a_i : i \in S\}$ and $A_D = \{a_i : i \in D\}$ we have
$eval_{static}(\mathbb{P}_S, A_S) \rightarrow \mathbb{Q}_{SA}$ and
$eval_{dynamic}(\mathbb{Q}_{SA}, A_D) \rightarrow b_A$

Simply put, 2-stage multi-staging refers to breaking down the execution of a program into two stages where the output of the first stage is a program specialized to a subset of the parameters. The generated program $\mathbb{Q}_{SA}$ is referred to as the partially evaluated program. In general, multi-staging can divide the execution into an arbitrary n-number of stages, with each stage specializing for 0 or more parameters. For most of this thesis, I will describe and use 2-stage multi-staging, but I will discuss extensions to support an arbitrary number of stages. Multi-Stage programming is also sometimes simply referred to as stage programming or staging.

2.1.1 Static and Dynamic Evaluation

The above definition of 2-stage multi-staging explicitly demarcates two separate execution or evaluation phases. We call these phases *static* and *dynamic* stages. The parameters $\{x_i : x \in S\}$ and $\{x_i : x \in D\}$ are respectively referred to as the *static* and *dynamic* parameters. The terms *static* and *dynamic* originate from the fact that the *static* arguments of a program do not change often, while the *dynamic* parameters change quite a lot. Thus, specializing the program for the *static* parameters often leads to performance improvements by amortizing the cost of the *static* evaluation phase. The differences in the implementation, differences in language choices for the *static* and the *dynamic* stage, and even differences in workflows lead to different styles of multi-staging with different benefits and drawbacks. Since the idea of multi-staging can be extended to have more stages by recursively expanding either of $eval_{static}$ or $eval_{dynamic}$ into more stages, the taxonomy introduced can independently apply to each such expansion. In the upcoming sections, I will try to summarize different styles of multi-staging, their benefits, and examples of systems that fit in those categories. By no means is the taxonomy below exhaustive, but this list covers the most popular forms of multi-staging and is most relevant to understanding the contributions of this thesis.

2.2 Multi-Staging: Homogeneous vs Heterogeneous

The easiest way of classifying multi-staging is based on the program representation used for $\mathbb{P}$, $\mathbb{P}_S$, and $\mathbb{Q}_{SA}$.

We call the multi-staging **Heterogeneous** if $\mathbb{P}_S$ and $\mathbb{Q}_{SA}$ are implemented or represented in different programming languages. Such a type of multi-staging is a very powerful way of combining the benefits of different languages, like performance, abstraction, and security properties. For example, high-level domain-specific languages like Lustre [31] or SCADE [38] that use a synchronous data-flow programming model but generate efficient imperative low-level C code or DSLs like GraphIt [201, 198, 25, 23] that represent programs using higher-level graph operations and generate low-level optimized C++ or CUDA. Heterogeneous multi-staging also allows for the separation of concerns or hardening the security of systems. For example, PHP or ASPX running on a web server generates HTML/JS/CSS to be rendered on the user's browser. While languages like PHP have good library support and interfaces to interact with server-side resources like databases, file systems, the generated code in HTML, JS, and CSS have first-class language support to interact with the browser and the DOM elements.

Homogeneous multi-staging on the other hand refers to implementations where $\mathbb{P}_S$ and $\mathbb{Q}_{SA}$ are implemented in the same language. Homogeneous multi-staging offers the inherent benefit of having similar syntax and semantics in the two stages, making it easier to move code and parameters between stages, and also offers interoperability between the code from the two stages. Homogeneous multi-staging also reduces the cognitive load on the developer since they are thinking of the static stage code and the generated code in the same language. Embedded DSLs in C++ printing out or generating optimized code in the same language, like Halide [146], Tiramisu [7] fit in this category.

2.2.1 Seamless Homogeneous Multi-Staging

Homogeneous multi-staging can further be divided into **Seamless** multi-staging and **Manual** multi-staging based on how close the representation of $\mathbb{P}_S$ is to the representation of $\mathbb{P}$ if $\mathbb{P}$ was written in the same languages as those of $\mathbb{P}_S$ and $\mathbb{Q}_{SA}$. In manual multi-staging, the developer has to use explicit constructs that assemble the program to be generated by either concatenating strings or calling constructors for program nodes, as opposed to seamless multi-staging, where the program $\mathbb{P}_S$ is represented in a very similar way to $\mathbb{P}$ with very few changes. The representation of $\mathbb{Q}_{SA}$ is irrelevant since it is generated and doesn't add to the cognitive load of the developer. Instead of being two separate categories, seamless vs manual multi-staging is a spectrum where multi-staging can be more seamless or less seamless than others. DSLs like Halide [146], Tiramisu [7] or TVM [35] or languages like PHP that have a specialized syntax to expression the computations are less seamless while languages like MetaOCaml, LISP/Scheme, Racket, Template Haskell that use quotation or quasi-quotation to generate code are more seamless since they require just adding a few annotations to parts of the program. Frameworks like PyTorch [138], TensorFlow [175], or Lightweight Modular Staging (LMS) [151], where the only change is to the type or annotations of the variables and the operations are extracted using operator overloading, are among the most seamless staging techniques. Seamless multi-staging further reduces the cognitive burden for the developer since they have to write code as if it were written to be executed in a single stage.

Figure 2.1 shows the implementation of a power function written in Scala with the LMS framework. Notice the code is similar to a regular Scala program with only the difference of the `Rep[]` type annotation. I will describe such a type-based approach in more detail next.

```
// power function: takes a staged base and an integer exponent
def power(base: Rep[Double], exponent: Int): Rep[Double] = {
    var result = unit(1.0)
    for (_ <- 0 until exponent) {
        result = result * base
    }
    result
}
```

Figure 2.1: Implementation of a power function in Scala written using the Lightweight Modular Staging Framework

Type-Based Multi-Staging

Type-Based Multi-Staging is a specific case of highly seamless multi-staging where the declared types of variables and expressions determine what stage they would be supplied and evaluated in. It is a form of seamless multi-staging where the program representation of $\mathbb{P}$ and $\mathbb{P}_S$ are the same Except for the declared types of the parameters and intermediates, which are changed to control whether they are evaluated in the *static* stage or the *dynamic* stage. Type-Based multi-staging was first introduced by the LMS System embedded in Scala [151] that uses `Rep[]` to defer execution of the expressions and statements to the *dynamic* stage.
Formally, we define two type-level functions - $\mathbb{D}$ and $\mathbb{S}$ which can be applied to any type T in the programming language of $\mathbb{P}$. The multi-stage program $\mathbb{P}_S$ is obtained by replacing the declared type of x_i, T_i with $\mathbb{S}(T_i)$ $\forall i \in S$ and replacing the declared type of x_i, T_i with $\mathbb{D}(T_i)$ $\forall i \in D$. Furthermore, the declared type of the return value y, T_y, is replaced with $\mathbb{D}(T_y)$. Finally, the declared type of each intermediate value I, T_I is replaced with $\mathbb{D}(T_I)$ if the computation of I depends on any value in $\{x_i : i \in D\}$ or another intermediate I' that has had its declared type replaced with $\mathbb{D}(T_{I'})$. The rest of the intermediates can have their type T_I be replaced by either $\mathbb{D}(T_I)$ or $\mathbb{S}(T_I)$. The freedom of choice for intermediates that don't depend on *dynamic* values allows the developer the choice on how much they want to specialize the program, offering a wide range of performance tradeoffs.

This form of seamless multi-staging is extremely powerful since, firstly, it allows converting any program $\mathbb{P}$ into its multi-stage variant $\mathbb{P}_S$ by simple type changes and without introducing any new constructs for creating program nodes or describing control flow or representing code as strings. Furthermore, moving code between stages becomes as simple as changing the declared type of variables and expressions without rewriting any code. The usability of such systems is further boosted by the fact that many popular languages like C++, Rust, Typescript, Swift, and Julia offer generics and type inference over declared types of variables, making it even more seamless. A simple consequence of such an approach is that if the type changes are reversed, which is often equivalent to dropping or ignoring some annotations, the behavior of the program would be equivalent to the original program $\mathbb{P}$.

2.3 Multi-Staging: Implicit vs Explicit

Another straightforward way of classifying multi-staging techniques is based on whether the evaluation of $eval_{static}$ on $\mathbb{P}_S$ and A_S produces the source code for $\mathbb{Q}_{SA}$ as an explicit output (to standard out or a file) or if it is just an intermediate produced inside the execution environment that can then be directly invoked as part of $eval_{dynamic}$. If the evaluation $\mathbb{P}_S$ produces the source code as output, we call it **Explicit** multi-staging. Languages like PHP, or DSLs like GraphIt, Halide, TACO fit in this category. Whereas if the execution environment of $eval_{static}$ and $eval_{dynamic}$ is connected in a way that the program just gets a handle to execute the specialized program $\mathbb{Q}_{SA}$, we call it **Implicit** multi-staging. Frameworks like PyTorch, TensorFlow, TemplateHaskell, and MetaOCaml fit in this category. C++ templates lie on the boundary of these two categories since, even though it produces a binary executable as part of the compiler (where the template evaluation executes), the output is an executable binary containing assembly instructions and not source code. Many frameworks also offer a mixed/hybrid mode where they allow directly producing a handle for execution, like in implicit multi-staging, but can also dump a readable description or source code with some auxiliary functions. Languages like Julia, C, and C++ preprocessor fit in this category. DSLs like TACO, which usually follow the explicit multi-staging route, also provide functions to automatically compile and load the generated code, making them mixed/hybrid mode.

Although implicit multi-staging is often easy to work with and integrate since it doesn't require adding an entirely new phase of compilation and execution, and the program that is generating the code can directly use the generated code, explicit multi-staging offers great benefits in terms of debuggability. Since the generated code can be manually inspected to fix bugs in the first stage code, instead of relying only on the final output for debugging. If the developer wishes, they can also further modify this generated code to fix bugs or performance issues if they are inclined to do so. Explicit multi-staging also offers the advantage that it makes implementing heterogeneous multi-staging easy. For languages that have incompatible runtimes or need to enforce separation of execution for safety or security reasons, explicit multistaging is the only viable technique.

2.4 Multi-Staging: Functional vs Imperative Perspective

Another way of classifying multi-stage programming techniques depends on the programming paradigm used for implementing the program $\mathbb{P}_S$. The choice of paradigm offers benefits and drawbacks both in terms of usability and performance, and also the implementation complexity of the evaluation procedure $eval_{static}$. Let us look at the differences in the two main paradigms: Functional and Imperative, which are the most contrasting in terms of the outcomes. Other paradigms, such as logic-based, task-based, or reflective programming, fit between these two extremes.

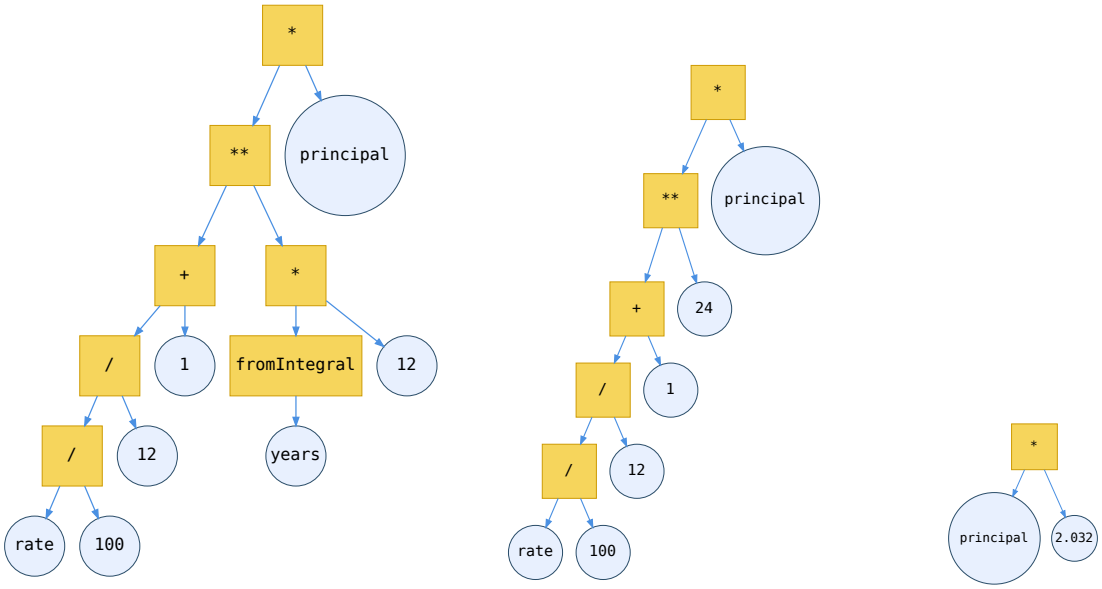
2.4.1 Functional Multi-Stage Programming

A functional programming paradigm is defined as a programming style that constructs entire programs by composing pure functions, immutable values, and higher-order functions. This style of programming is characterized by a lack of assignments to variables and any control flow constructs like loops and conditions. The functional programming paradigm can be used in most languages,

regardless of whether they support mutations; however, many specialized functional programming languages like Haskell, OCaml, Elm, or Lisp have been developed to facilitate writing complex functional programs. Due to the lack of mutations and being made from a composition of pure functions, programs in the functional programming paradigm can be represented as a tree of values (or DAG) with the return value at the root and the program inputs and constants at the leaves. However, practical functional programming systems often include controlled side effects (monads, exception handling, I/O) or laziness that may appear to deviate from this representation, but the core idea that there are direct edges from the inputs to the output still stands. Figure 2.2 shows a program in Haskell to compute the compound interest given the principal amount, rate of interest, and the number of years, and Figure 2.3a shows the value tree representation of the same.

```haskell
compoundInterest :: Float -> Float -> Int -> Float
compoundInterest principal rate years =
  principal * (1 + rate / 100 / 12) ** (12 * fromIntegral years)
```

Figure 2.2: Implementation of a pure function in Haskell that accepts the principal amount, rate of interest, and time in years and returns the compound interest accrued.



(a) (b) (c)

Figure 2.3: a) Value-tree representation of the `compoundInterest` function defined in Figure 2.2. Program inputs and constants are shown as blue squares, and the operations are shown in square boxes. b) Program value-tree specialized for `years = 2`. c) Program value-tree specialized for `years = 2` and `rate = 36`.

The main observation here is that in the functional paradigm, there is an explicit edge in this representation between any value and all the values/inputs it depends on. Even though this program

just has one unconditional expression, programs with control operators like conditionals, recursive function calls, and repeated operations can be represented similarly with higher-order function calls like `foldl`, which just become part of the expression tree. When combining multi-stage programming with a functional paradigm, the process of *static* stage evaluation can be thought of as trimming of subtrees from the value-tree and replacing it with a computed value. Such a transformation can only be done if all the leaves of the subtree are either constants or a parameter $\in \{x_i : i \in S\}$. This is obvious because if the subtree depends on a parameter $\in \{x_i : i \in D\}$, the resulting value of the subtree cannot be computed with the inputs known at the *static* stage. Figure 2.3b shows the program tree for $\mathbb{Q}_{SA}$ for the static inputs `year = 2`. Notice the subtree for `12 * fromIntegral years` is replaced with just `24`. Further, Figure 2.3c shows the program tree for $\mathbb{Q}_{SA}$ for the static inputs `year = 2, rate = 36`. The whole subtree for the compound factor value is replaced with the resultant value `2.032`. As a side note, this example provides an intuition for why a specialized version $\mathbb{Q}_{SA}$ is usually faster than the original program $\mathbb{P}$ since it has fewer operations. Figure 2.4 and Figure 2.5 show the corresponding specialized programs.

```
1 compoundInterest :: Float -> Float -> Float
2 compoundInterest principal rate =
3   principal * (1 + rate / 100 / 12) ** 24
```

Figure 2.4: Specialized version ($\mathbb{Q}_{SA}$) of the program in Figure 2.2 for inputs `years = 2`.

```
1 compoundInterest :: Float -> Float
2 compoundInterest principal =
3   principal * 2.032
```

Figure 2.5: Specialized version ($\mathbb{Q}_{SA}$) of the program in Figure 2.2 for inputs `years = 2, rate = 36`.

Implementing Functional Multi-Staging

Due to the simplicity of the process of partially evaluating programs in the functional paradigm, the implementation of the $eval_{static}$ evaluation procedure is also straightforward. Essentially, the evaluation procedure $eval_{static}$ can reuse the evaluation procedure of the host language with minor extensions. Since the evaluation of the subtree based on *static* inputs is just like the regular evaluation of a program in the language, no special constructs are required. The program, dependent on the *dynamic* inputs, however, needs to be generated, which can be done by introducing a new syntax to quote parts of the program to be evaluated later. Languages like MetaOCaml and Haskell use this approach. The return type of any subtree containing a quoted subtree is changed to be a program type, and as it propagates up, the return type of the whole program is a program. Regular execution of the program $\mathbb{P}_S$ with the arguments $\{a_i : i \in S\}$ now returns $\mathbb{Q}_{SA}$ without requiring any special execution mechanisms. By just changing the behavior of operations that deal with program types, the tree pruning can be implemented cleanly.

Figure 2.6 shows the multi-stage version of the compound interest program in Figure 2.2 written in TemplateHaskell. Notice the special syntax with `[|` and `|]` to quote parts of the program. The dynamic parameters are supplied in the quoted block, and the return of the whole function is changed to `Q Exp`, which is a program type in TemplateHaskell. The code snippet also shows the invocation of this function for `years = 2`, which is the first stage execution using the evaluation procedure $eval_{static}$ to produce a specialized program. The specialized program $\mathbb{Q}_{SA}$ is not visible here and is a program object that can be called simple TemplateHaskell uses implicit multi-staging. Multi-Stage systems like Lightweight Modular Staging (LMS) [151] when used to stage functional programs don't even need specialized quoting syntax and just use the declared types of variables and polymorphism to identify which part of the tree should be pruned and which should be kept.

Thus, the functional programming paradigm is a great fit for multi-stage execution since it can reuse the majority part of the execution machinery to implement the programming model for $\mathbb{P}_S$ and $eval_{static}$. However, as explained in Chapter 1, a major reason for multi-stage execution is to produce more optimized code. Generating high-performance code requires exercising control over the way and order in which computations in the program are evaluated. Since all the above examples generate programs in the functional paradigm, such control is often not possible, and the applicability of multi-staging in functional languages for high-performance remains limited. This is evident from the fact that most high-performance code running on CPUs or GPUs is written in languages like C, C++, Rust, or CUDA, which are inherently imperative. Furthermore, DSLs with a focus on high-performance also generate programs in these languages. It is imperative that the ideas of multi-staging be brought to the imperative world in the most seamless way possible.

```
1  genCompoundInterest :: Int -> Q Exp
2  genCompoundInterest years =
3    [| \principal rate ->
4         principal * (1 + rate / 100 / 12) ** (12 * fromIntegral years)
5     |]
6
7  compoundInterest2 :: Float -> Float -> Float
8  compoundInterest2 = $(genCompoundInterest 2)
```

Figure 2.6: A staged version of the program in Figure 2.2 in TemplateHaskell for specializing the function for the parameter `years`.

2.4.2 Imperative Multi-Stage Programming

The Imperative programming paradigm is characterized by two main features - mutations and data-dependent control flow. Imperative programs are composed of variables that can be assigned and reassigned values over the course of execution of the program. Besides values and expressions, these programs also contain statements that are to be executed in a fixed order. Statements can be nested under control flow constructs like if-then-else conditions and loops, which change the number of times statements are executed. Since the order of evaluation of a statement is important, unlike functional programming, the control-flow constructs add implicit dependency to the output

that is not always obvious in the program representation. Figure 2.7 shows a simple program written in Python to compute the power function given a base and exponent. The types of values have been annotated.

```
def power (base: Float, exp: Int):
    res = 1
    for e in range(exp):
        res *= base
    return res
```

Figure 2.7: Implementation of a power function in Python given base and exponent

Right away, it is clear that programs written in imperative style cannot be converted into program trees but rather a recursively nested set of statements that then contain expressions and value trees. The problem is that values assigned in the program depend not only on the variables that appear on the right-hand side of the assignment but also implicitly depend on the values that control the conditions and loops these assignments are part of. In the power program shown in Figure 2.7, the return value **res** is only ever assigned with expressions based on constants, itself, and **base**. However, the true value of the result depends on how many times the loop runs, which in turn depends on the value of **exp**. Thus, there is an implicit dependency between **ret** and **exp**. If we were to apply multi-staging to this program where **exp** is a static input, this dependency must be made explicit, and the program would have to be transformed in ways to reflect that relationship. This dependency becomes even more complicated when the control flow depends on one of the *dynamic* parameters; the evaluation cannot simply resolve the dependency during $eval_{static}$ evaluation.
The evaluation of every expression needs to look at the control-flow constructs it is a part of to gather the implicit dependencies, and this cannot be achieved without inspecting the program globally during the $eval_{static}$ procedure. In conclusion, implementation of multi-staging in imperative languages is significantly harder than in functional languages and may require more intrusive techniques than simple changes to *eval* to obtain $eval_{static}$.

2.5 Imperative Multi-Staging Implementation Approaches

It is clear that implementing multi-staging in imperative requires analyzing the whole program and the context around the expression to capture the true dependencies between values. The program transformation procedures in $eval_{static}$ thus also require access to operations that might typically not be available in the host language. Furthermore, imperative programming is the most helpful in exercising maximum control over the low-level operations required to obtain the performance, and hence bringing multi-staging to imperative languages is key to high-performance DSL development. We will look at various approaches taken by different imperative multi-staging systems and understand how expressive and seamless they are.

2.5.1 Compiler-Based Multi-Staging

The implementation of $eval_{static}$ requires access to the context in which a statement is executed. Since in most popular programming languages like C, C++, or Python, this information is not available within the expression itself, we need to move the execution of $eval_{static}$ where the program representation is available and can be inspected. An obvious solution is to make $eval_{static}$ part of the compilation step, where the program representation is available as ASTs or program IRs. This is the approach taken by C++ templates, Rust's proc macros, or DSLs embedded in Python that rely on inspecting ASTs. These languages, once again, extend the syntax of the language to allow separating the *static* code from the *dynamic* code and extend the compilation infrastructure to evaluate the *static* stage with the *static* inputs to produce $\mathbb{Q}_{SA}$ directly.

Figure 2.8 shows the block diagram for how the compiler infrastructure for a language like C++ is extended to support template metaprogramming-based multi-staging. The C++ source code with template language constructs is fed into the extended compiler. The compiler now needs extensions to the parser to support the new keywords and new language syntax. The extracted program AST is now stored as an AST augmented to represent templates. The crucial step of this compiler, shown in red, is a template evaluation engine that performs the execution of the template program with the template parameters (baked into the program itself) to produce a regular C++ IR that can be passed off to the regular C++ compilation pipeline. The template evaluation engine thus essentially implements all the capabilities of $eval_{static}$, which means that features available during $eval_{static}$ are bottlenecked by features present in the template evaluation engine. Since the compiler has to guarantee behavior like never crashing due to an invalid access or integer overflow or termination guarantees, the capabilities of the template evaluation engine remain fundamentally limited.

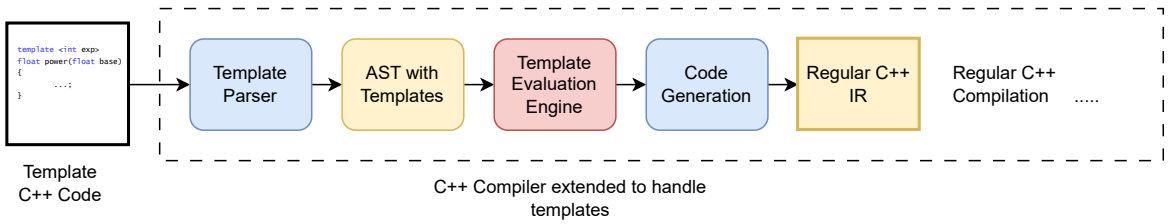


Figure 2.8: Block diagram showing extensions to a regular C++ compiler to add support for evaluating templates.

The reason developers prefer C++ for high-performance and other domains alike is because of the rich set of features that make the language expressive and fit for exactly describing how the computation should be performed. C++ is a multi-paradigm language with features like mutations, dynamic memory allocation, pointer indirection, object-oriented programming, and clean abstractions for performing I/O. Templates, on the other hand, have access to neither of the above, severely restricting the programming model for $eval_{static}$. Some of these features have been added to new C++ versions with the `constexpr` and `consteval` keywords that allow compile-time execution with some dynamic memory allocations and mutations; however, the capabilities remain extremely limited, and the users have to use a different syntax. Due to all these limitations, C++ compile-time evaluation support scores low on the seamless multi-staging axis. Finally, since the code for the

template evaluation is running inside the compiler, support for external libraries is not possible since it would require extending the compiler. All of these limitations motivate the need for building a multi-staging system where $eval_{static}$ has the same features as $eval_{dynamic}$ or $eval$.

2.5.2 Execution-Based multi-staging

The lack of features in most compiler-based multi-staging systems urged developers to move towards systems that reuse existing language features to implement the first stage execution. This is typically done by embedding a new language in the host language by creating a library abstraction that allows the developers to write $\mathbb{P}_S$. We call this Execution-Based multi-staging since it reuses the execution engine of the host language. These systems also typically take the homogeneous multi-staging path, making the set of features available across all stages the same. Formally, the evaluation procedure $eval_{static}$ is exactly the same as $eval_{dynamic}$, providing a consistent experience. Just like functional multi-staging, the abstraction transforms the program $\mathbb{P}$ into a $\mathbb{P}_S$ which produces $\mathbb{Q}_{SA}$ simply by executing it. This approach is taken by systems like Tensorflow [175, 3], where polymorphism and operator overloading on the type of variables are used to execute the programs with a mix of concrete and placeholder values that track operations performed on them.

TensorFlow introduces the **`tf.placeholder`** class. For transforming $\mathbb{P}$ into $\mathbb{P}_\mathbb{S}$ all the static parameters $\{x_i : i \in S\}$ are bound to the respective arguments $\{a_i : i \in S\}$. The dynamic parameters $x_i : i \in D$ are bound to new objects of type **`tf.placeholder`**. The regular values in the program execute as it is, while the **`tf.placeholder`** objects that do not have any concrete values simply log all operations being performed and return a trace of operations as the final value. The execution of $\mathbb{P}_S$ as a program in Python thus yields a program that consists of a sequence of operations on the dynamic parameters. Figure 2.9 shows a simple program written with the Tensorflow library that adds three parameters **`a, b, and c`** and specializes the program for **`c = 42`**. Upon execution, **`sum2`** holds a TensorFlow expression object equivalent to the program **`a + b + 42`** that can be called with concrete values for **`a`** and **`b`**.

```
1 import tensorflow as tf
2
3 def sum3(a, b, c):
4     return a + b + c
5
6 x = tf.placeholder(tf.int, "a")
7 y = tf.placeholder(tf.int, "b")
8 z = 42
9 sum2 = sum3(x, y, z)
```

Figure 2.9: A simple program in the Tensorflow library in Python for summing 3 variables. The program is invoked with 2 placeholder values and one concrete input.

As explained before, the type **`tf.placeholder`** does not have a concrete value, but all the operations like **`__add__`**, **`__sub__`**, **`__mul__`**, etc, are overloaded to recursively produce equivalent operations to be returned. This is similar to how expression trees are handled in functional multi-staging.

```
import tensorflow as tf

def max (a, b):
    m = tf.cond(a > b,
        lambda: a, # then branch
        lambda: b) # else branch
    return m

def sumn(n):
    sum = tf.constant(0)
    sum, _ = tf.while(lambda n, _: n > 0, # loop condition
                  lambda n, sum: (n - 1, sum + n), # body and new values
                  [n, sum]) # initial values
    return sum

max(tf.placeholder(tf.int, "a"), tf.placeholder(tf.int, "b"))
sumn(tf.placeholder(tf.int, "n"))
```

Figure 2.10: Using control-flow constructs on tf.placeholder values in Tensorflow with the **`tf.cond`** and **`tf.while`** functions.

Besides expression, there are two types of control flow to handle. The control-flow based on purely *static* parameters remains unchanged and executes like usual. Since *static* parameters have concrete values, conditions and loops can be completely resolved, and the appropriate branches are selected, and the bodies of loops are unrolled an appropriate number of times. However, control-flow constructs where the execution depends on at least one *dynamic* parameter cannot be fully evaluated due to the lack of concrete values and must be deferred to the evaluation of $\mathbb{Q}_{SA}$ like the binary operations on the placeholder values. However, since most languages do not allow overloading control-flow operators in a meaningful way, extracting these control-flow constructs as program objects is challenging. Developers of most execution-based multi-staging systems side-step these issues by either completely disallowing control-flow on *dynamic* parameters (C++ templates and constexpr) or by providing replacement control-flow syntax that generates the condition or loop instead of executing it. Figure 2.10 shows how loops and **`if-then-else`** conditions are written on Tensorflow's placeholder values for a function **`max`** that returns the greater of two inputs and **`sumn`** that returns the sum of the first **`n`** integers. The library introduces two new functions **`tf.cond`** and **`tf.while`** for creating if-then-else conditions and while loops, respectively. These functions accept as their first parameter an expression object that represents the condition in the if-then-else or the loop. The **`tf.cond`** accepts two lambdas corresponding to the then and the else branch. These lambdas return the values for certain placeholder values if that branch was taken. Similarly, the **`tf.while`** accepts a single lambda which corresponds to the body of the loop and returns the new values for the placeholder after one iteration. Both these operators return the final value of the relevant placeholder values. Essentially, the control-flow constructs are converted into functional objects that create a program with loops and conditions in them.

Figure 2.11 shows the implementation of the **`power`** function using repeated squaring in Python

```
import tensorflow as tf

def power(base, exp):
    res = 1, x = base
    while exp > 0:
        if exp % 2 == 1:
            res = res * x
        x = x * x # Repeated squaring for power
        exp = exp / 2
    return res

def gen_power(base, exp):
    res = tf.constant(1), x = base
    res, _, _ = tf.while (lambda _, _, exp: exp > 0, # condition for termination
                          lambda res, x, exp: (tf.cond(exp % 2 == 1, res * x, res),
                                               x * x, exp / 2), # updated values
                          [res, x, exp] # initial values
    return res

power_staged = gen_power(tf.placeholder(tf.int, "base"), tf.placeholder(tf.int, "exp"))
```

Figure 2.11: Implementation of the power function that returns base power exponent using repeated squaring in Python and in Tensorflow. The implementation shows the transformations required to convert conditions and loops into functional objects.

and the corresponding version in Tensorflow. The power function contains a loop that is dependent on the exponent and a condition inside the loop based on whether the current value of the exponent is odd or even. The **`gen_power`** function, on the other hand, has nested calls to **`tf.while`** and **`tf.cond`**. Notice that the changes required aren't just local replacements but require writing the program in a functional way. While this approach solves the problem, it greatly sacrifices seamlessness while transforming $\mathbb{P}$ into $\mathbb{P}_S$.

2.5.3 Hybrid multi-staging

The lack of features in compiler-based multi-staging and the reduced seamlessness in execution-based multi-staging due to the lack of overloading for control flow in most host languages prompts the need for a hybrid approach that combines the benefits of both approaches. Hybrid multi-staging or compiler-assisted execution-based multi-staging is the idea where the generation of the resultant program is done using the capabilities of the host language, just like in execution-based multi-staging; however, the natural control-flow operators like **`if-then-else`** and **`while`** loops are converted to special functions using a compiler pass or plugin. The execution procedure $eval_{static}$ is broken down into two separate steps. The program $\mathbb{P}_S$ is first passed through a compiler tool that replaces all control-flow dependent on *dynamic* values into special function calls that produce those functions, like in execution-based multi-staging to produce an intermediate program $\mathbb{P}'_S$. We call this conversion function $preprocess$. The generated intermediate program is then evaluated using the evaluation procedure of the host language $eval$, like in execution-based multi-staging to produce $\mathbb{Q}_{SA}$. Formally:

$$preprocess(\mathbb{P}_S) \rightarrow \mathbb{P'}_S$$
$$eval_{static}(\mathbb{P}_S, A) = eval(preprocess(\mathbb{P}), A) \rightarrow \mathbb{Q}_{SA}$$

This approach is taken by frameworks like Lightweight Modular Staging (LMS) [151], which is a type-based multi-staging framework embedded in Scala. Being a type-based multi-staging system, LMS is extremely seamless. The type-level functions $\mathbb{D}$ and $\mathbb{S}$ are simply defined as - $\mathbb{D} = T \rightarrow \textbf{Rep}[T]$ and $\mathbb{S} = T \rightarrow T$. With these definitions, the *static* parameters and the intermediates solely depending on them are left as it is, while the *dynamic* parameters have their types T replaced with $\textbf{Rep}[T]$, where **Rep** is a keyword introduced by $preprocess$. The LMS-enabled Scala Compiler runs the $preprocess$ step implicitly to produce $\mathbb{P'}_S$ where all conditions on expressions of type $\textbf{Rep}[T]$ are replaced with calls to the runtime library function `cond_if` (similar to `tf.cond`) and while loops on such expressions are converted to calls to `while_loop` (similar to `tf.while`). This new program is then compiled to produce an executable that runs with inputs $\{a_i, i \in S\}$ to produce $\mathbb{Q}_{SA}$. Figure 2.12 shows the three programs $\mathbb{P}_S$, $\mathbb{P'}_S$ and $\mathbb{Q}_{SA}$. As shown in the code snippet, the binary operators are also replaced here to better handle overloading on $\textbf{Rep}[T]$ types, but the operations on non-$\textbf{Rep}[T]$ types are left the same. In this figure, the program $\mathbb{Q}_{SA}$ is specialized to have the statement `z = z + 1` unconditionally.

```
1 val x: Rep[Int]
2 val y: Rep[Int]
3 val z = x + y
4 // Regular Int (static)
5 val w: Int = 13
6 if (w > 2) z = z + 1
7 if (x < y) {...}
```

(a) $\mathbb{P}_S$

```
1 val x: Rep[Int]
2 val y: Rep[Int]
3 val z = _infix_plus(x, y)
4 val w: Int = 13
5 // nonRep Operations unchanged
6 if (w > 2) z = _infix_plus(z, 1)
7 cond_if((x < y), []{...})
```

(b) $\mathbb{P'}_S$

```
1 val x: Int
2 val y: Int
3 val z = x + y
4 // nonRep condition
5 // evaluated
6 z = z + 1
7 if (x < y) {...}
```

(c) $\mathbb{Q}_{SA}$

Figure 2.12: The three programs $\mathbb{P}_S$, $\mathbb{P'}_S$ and $\mathbb{Q}_{SA}$ in LMS

Since the compiler is only performing the preprocessing and the actual program generation is performed by the host language *eval* execution procedure, the features available during the *static* stage are not limited by any additional constraints. Furthermore, as seen in Figure 2.12a, the program is very similar to the hypothetical Scala program $\mathbb{P}$, except for the types; the whole system is extremely seamless. Thus, Hybrid multi-staging combines the benefits of both compiler-based multi-staging and execution-based multi-staging. Consequently, systems like LMS [151] are some of the most popular choices for multi-stage systems and have been used to stage many complex systems with ease.

However, all the above approaches that use a specialized construct for control-flow directly or indirectly and try to generate control flow at a syntactic level suffer from the curse of the Side-effects Leak Problem described in the next section.

2.6 Side-effects Leak Problem

The different techniques described in Section 2.5 have different benefits and drawbacks in terms of programming features available during staging or the seamlessness of writing a staged program. These different characteristics may also have an effect on the quality of the code generated and its performance. However, all the above-described techniques handle control-flow constructs on *dynamic* inputs in a similar way, directly or indirectly. Compiler-based, execution-based, or hybrid multi-staging all handle control-flow statements like if-then-else and while loops at a syntactic level. Meaning either the compiler or the developer identifies these constructs in the source code (or program AST) and replaces them with special function calls that accept the body of these statements as closures. During the execution of $\mathbb{P}_S$ to generate $\mathbb{Q}_{SA}$, whenever these functions are called, the implementation of these functions separately evaluates the body of these statements by invoking the closures passed in isolation. The evaluation of these closures provides the body of the loop or the branches in the if-then-else, which are then stitched into the main program by creating the appropriate control-flow construct. Thus, for every if-then-else in the original source code, one if-then-else is produced in the generated code, and similarly for loops. We also notice that in the case of if-then-else, both the then branch and the else branch are evaluated one after the other, regardless of which branch would actually be taken during the execution of $\mathbb{Q}_{SA}$ with the *dynamic* inputs. This is because during the *static* evaluation phase, no concrete values for *dynamic* values are known. Since the programming paradigm we are looking at is imperative, the code can have side effects. This evaluation of both branches of the if-then-else can thus lead to improper side-effects on the first stage or *static* intermediates violating program correctness.

Let us look at the concrete example shown in Figure 2.13. The program is written in C++. In this example, the *dynamic* inputs, `y` and `z`, are shown in green, while the *static* input `x` is shown in blue. The simple program shows an if-then-else condition. Since the condition `(y > 5)` is dependent on a dynamic parameter, a special `if_then_else` function is inserted (either by the developer or by the compiler during $preprocess$). This function accepts the condition and the two C++ lambdas corresponding to the then and the else branch. Suppose this program is executed with *static* inputs `x = 5` and the `if_then_else` function is called, the function first evaluates `(y > 5)` to obtain the program representation for the expression of the condition. It then invokes the then lambda. This invocation returns the code `z += 5`, since `5` is the current concrete value of `x`. At the same time, during the invocation of the lambda `x` is incremented by `1`, setting its new value to be `6`. Now the function invokes the else lambda and extracts the else body `z -= 6`. However, since the then-lambda was just executed, `x` has already been mutated and the wrong constant is embedded in the generated else branch. In the unstaged program $\mathbb{P}$ `z` is only supposed to be decremented by `5` since the then branch wouldn't be evaluated if the else branch is to be evaluated.

The core of the problem here is not isolating the side-effects on *static* parameters and intermediates when the updates are guarded by conditions on *dynamic* expressions. This problem is even worse with loops, where similarly, if there are side-effects on *static* variables inside loops where the trip-count is dependent on a *dynamic* input, evaluating the body just once might not produce correct code. Figure 2.14 shows this case with the `while_loop` function call. Here once again the parameter `x` is *static* and `y` and `z` are *dynamic*. The loop created using the call to `while_loop` has a condition based on `y`. The body of the loop accumulates `x` in `z` and increments `x` by `1`. The variable `y` is

```
int foo (int x, int y, int z) { // x is static and y, z, and return values are dynamic
    if_then_else((y > 5,
        []() {
            z += x;
            x = x + 1;
        },
        []() {
            z -= x;
        });
    return z;
}
```

Figure 2.13: A program demonstrating the side-effect leak problem due to mutations on *static* variables under conditions based on *dynamic* values. Dynamic inputs are shown in green.

decremented by `1` before the loop is repeated. If this program is executed with *static* arguments `x = 5`, the generated code would simply look as shown in Figure 2.15. Since the implementation of `while_loop` executes the body exactly once, the generated body would simply have the statement `z += 5`. The generated program now won't behave correctly since the accumulated value is supposed to increase in every iteration.

```
int foo (int x, int y, int z) { // x is static and y, z, and return values are dynamic
    while_loop((y > 0,
        []() {
            z += x;
            x = x + 1;
            y--;
        });
    return z;
}
```

Figure 2.14: A program demonstrating the side-effect leak problem due to mutations on *static* variables in the body of a loop where the trip count is dependent on *dynamic* values. *dynamic* parameters are shown in green.

This problem again happens because the side-effects on `x` are evaluated only once; however, the body is executed multiple times during the *dynamic* evaluation. Many systems, including C++ consteval, side-step this problem by simply disallowing side-effects on *static* variables inside control-flow based on *dynamic* inputs. While a viable solution, such side-effects leak problems arise a lot in complex code bases, especially while staging programs like interpreters, as we will see in Chapter 6, thus greatly restricting the kind of programs that can be multi-stage evaluated without lots of major refactoring. To solve this problem, the side effects on variables should be isolated to just the corresponding branch of execution. Thus, just a syntactic local handling of control-flow constructs is not a feasible approach for a correct implementation of imperative multi-staging.

```
int foo (int y, int z) {
    while (y > 0) {
        z += 5;
        y--;
    }
    return z;
}
```

Figure 2.15: Output of the program in Figure 2.13 for inputs `x = 5`. Naively generating the loop doesn't capture the effects of side-effects on *static* variables.

2.7 Multi-Staging Frameworks Comparison

So far, we have discussed how a variety of multi-staging frameworks can be classified into different types of multi-staging- Homogeneous vs Heterogeneous, varying degree of Seamlesness, Implicit vs Explicit, Functional vs Imperative, and Compiler-Based, Execution-Based or Hybrid multi-staging approach to implementation. I have also mentioned a few examples of frameworks from each category. We also discussed how most multi-staging framework implementations for imperative languages are susceptible to the side-effect leak problem. Before we go further and propose a new methodology for implementing multi-staging in a way that solves the side-effect leak problem, let us look at some of the most popular multi-staging frameworks, what category of the taxonomy they lie in, and if they are susceptible to side-effect leaks. Table 2.1 shows a list of multi-staging frameworks, and for each we mark if it is Heterogeneous/Homogeneous, Implicit/Explicit, Purely-Functional/Imperative, Compiler-Based/Execution-Based/Hybrid, and if it is free from side-effect leak problems. We also rank each by the degree of seamlessness they have. We notice that frameworks like Node.js, PyTorch, Tensorflow 2.0, and LMS rank high on seamlessness since they don't need any new syntax for expressions of control-flow constructs on *dynamic* types. Frameworks like TemplateHaskell, Scheme, MetaOCaml, and others that use quasi-quotation rank medium because they use the same syntax, but the *dynamic* code needs to be added to separate quoted blocks. The rest of the frameworks, like C++ templates or Tensorflow 1.0, introduce an entirely different syntax for the one of the stages and hence rank low. Similarly, most frameworks that are in purely functional languages do not suffer from the side-effect leakage problem because they don't allow side effects or control dependencies to begin with. Some other frameworks do not allow side-effects on *static* variables under conditions based on *dynamic* values. Hence, both are marked *N/A*. Frameworks like LMS or TensorFlow that suffer from problems due to not handling side effects correctly are marked so.

I also mention the BuildIt system [19], which is one of the main contributions of this thesis, to compare it to other systems. It is the only system that provides an imperative programming model, handles the side-effect leakage problem correctly, and ranks high on seamlessness because it takes a type-based approach. BuildIt does all this without making a single compiler change and uses a purely execution-based approach. In the next section, I will describe **REMS** or Re-Execution Based Multi-Staging that is the key to solving the side-effects leakage problem.

Framework	Homogeneous /Heterogenous	Implicit /Explicit	Functional /Imperative	Implement Strategy	Side-Effects Leakage Free	Seamlessness
PHP Server [2]	Heterogeneous	Explicit	Imperative	Execution	✗	Low
Node.JS	Homogeneous	Explicit	Imperative	Execution	✗	High
TemplateHaskell [156]	Homogeneous	Implicit	Functional	Hybrid	N/A	Medium
MetaOCaml [29]	Homogeneous	Implicit	Functional	Hybrid	N/A	Medium
Scheme [163]	Homogeneous	Dual	Imperative	Execution	✗	Medium
C++ Templates	Homogeneous	Implicit	Mixed	Compiler	N/A	Low
C++ consteval	Homogeneous	Implicit	Imperative	Compiler	N/A	Medium
pytorch [138]	Homogeneous	Implicit	Imperative	Execution	✗	High
Tensorflow 1.0 [175]	Homogeneous	Implicit	Mixed	Execution	✗	Medium
Tensorflow 2.0	Homogeneous	Implicit	Imperative	Compiler	✗	High
Halide DSL [146]	Heterogeneous	Explicit	Functional	Execution	N/A	Low
Terra [55]	Homogeneous	Explicit	Imperative	Hybrid	✗	Medium
LMS [151]	Homogeneous	Explicit	Imperative	Hybrid	✗	High
BuildIt [19]	**Homogeneous**	**Dual**	**Imperative**	**Execution**	✓	**High**

Table 2.1: List of different multi-staging frameworks and their categorization into different-types of multi-staging

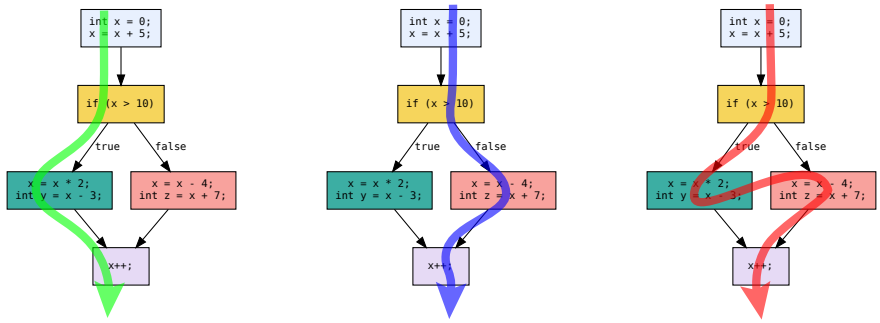
2.8 Re-Execution Based Multi-Staging (REMS)

In this section, I will introduce a novel methodology for implementing multi-staging in an imperative language without suffering from the side-effect leak problem described above.

Most compiler and execution based strategies rely on identifying the presence and extent of control flow based on the representation of $\mathbb{P}_S$ to create the control flow in $\mathbb{Q}_{SA}$. We call this syntax-based control-flow extraction. While these techniques provide high fidelity to the original implementation by generating code for $\mathbb{Q}_{SA}$ that is very similar to the original code, they also severely limit the kind of complex control-flow generated that is required for performance and, in some extreme cases, for correctness as seen in the side-effect leak problem.

The precise problem with syntax-based techniques is that the side effects of updates to variables are not restricted to the branch they are executed in. This is, in turn, because the order of execution of blocks of code is changed from the correct order of execution to aid with the extraction process. Figure 2.16b and Figure 2.16c show the two possible orders of execution of blocks for the program in Figure 2.16a with an if-then-else condition, while Figure 2.16d shows the wrong order executed while extracting this condition using a syntax-based extraction technique. Notice that the executed path does not correspond to any of the correct paths and thus violates the semantics of the programs. In most programs where side-effects are only on the *dynamic* variables, this change in order does not affect the behavior. However, with side-effects on *static* variables, this causes correctness issues. In fact, just violating the order of execution of the program may lead to undefined behavior in the program, in which case even a successful termination is not guaranteed. In the presence of multiple if-then-else conditions and loops, the execution of the program diverges quickly from the expected program.

```
1  int x = 0;
2  x = x + 5;
3  if (x > 10) {
4      x = x * 2;
5      int y = x - 3;
6  } else {
7      x = x - 4;
8      int z = x + 7;
9  }
10 x++;
```



(a) (b) (c) (d)

Figure 2.16: a) Program in C++ with an if-then-else condition. b) Possible order of execution for blocks in Figure 2.16a when the then-branch is taken. c) Possible order of execution for blocks in Figure 2.16a when the else-branch is taken. d) Wrong order of execution for the blocks when syntax-based multi-stage execution is used.

2.8.1 Control-Flow Exploration with Re-execution

The problem described above associated with syntax-based control-flow extraction techniques is rooted in the fact that the execution of the program diverges from any of the expected execution of the program, leading to undefined or incorrect behavior.

To ensure correctness, I will first define the notion of a **semantically-correct execution order**: A semantically-correct execution order is an execution of the statements in the program $\mathbb{P}_S$ where the order of the execution of the statements (composed of both *static* and *dynamic* variables) corresponds to any one of the valid orders of execution of the statements in the program $\mathbb{P}$ written in the same language and representation

A semantically-correct execution order guarantees all side-effects and reads on both *static* and *dynamic* happen in an order that is consistent with one of the executions in the original program. Since all the execution orders in the original program are correct and do not invoke undefined behavior, we can guarantee that an extraction methodology that solely relies on a semantically correct execution order does not violate program behavior and thus avoids undefined or incorrect behavior.

However, if we look carefully at Figure 2.16b and Figure 2.16c, we notice that neither of the paths highlighted in green or blue contains the complete information required to construct the program $\mathbb{Q}_{SA}$. The green path misses the statements in the else block, and the blue path misses the information in the then block. However, combined these two semantically-correct executions contain all the statements that need to be included in the generated program $\mathbb{Q}_{SA}$. Meaning a single semantically-correct execution guarantees a correct view of the program, but does not guarantee a complete view of the program. This means that to extract the complete program without violating program correctness, multiple executions are required.

With this observation, I will propose an execution-based control-flow extraction in imperative multi-stage code using only multiple semantically-correct executions. I will describe a procedure for converting $\mathbb{P}$ into $\mathbb{P}_S$ and the evaluation procedure $eval_{static}$. In the later Chapter 4, I will describe how these changes can be done in a feature-rich language like C++ without requiring any changes to

$eval_{static}$ using operator-overloading and other forms of polymorphism.

I have already explained the notion of *static* parameters and intermediates and *dynamic* parameters and intermediates. With this, the program representation for $\mathbb{P}_S$ is exactly the same as that of $\mathbb{P}$ with added annotations to identify *static* parameters and intermediate variables and *dynamic* parameters and intermediates. The annotations of parameters to the program are decided solely based on the set S such that a parameter x_i is annotated as *static* if $i \in S$ and x_i is annotated as *dynamic* if $i \in D$. Furthermore, all intermediate program variables that have initialization or assignments based on expressions that depend on at least one variable annotated as *dynamic* are annotated as *dynamic*. The rest of the intermediate variables can be annotated as *static* or *dynamic*, but are all annotated as *static* for maximum specialization.

The evaluation procedure for all operations and assignments on variables annotated *static* is exactly the same as in the evaluation procedure *eval* of the host language. The evaluation procedure for if-then-else and loops where the conditions depend on expressions composed only of *static* variables is also exactly the same as in the evaluation procedure *eval*. This means the variables annotated *static* have concrete values during the evaluation of $\mathbb{P}_S$ and are read and written as regular variables. The evaluation procedure for variables annotated *dynamic* is modified to treat them as placeholder values. Each variable is assigned a unique name at variable creation. The evaluation of operations on these variables and expressions creates equivalent program nodes to be generated as part of $\mathbb{Q}_{SA}$. The leaf nodes of these program expression trees are either the names for the placeholders or constants which are obtained when *static* values are operated with *dynamic* values. The value of the constant leaf-node is the same as the concrete value of the *static* expression at the point of execution. Similarly, assignments and statements with *dynamic* values produce program nodes to be generated as part of $\mathbb{Q}_{SA}$.

With this implementation of the evaluation procedure $eval_{static}$, it is easy to follow how the evaluation of programs $\mathbb{P}_S$ containing straight-line code would produce the equivalent partially evaluated program $\mathbb{Q}_{SA}$. This procedure is exactly the same as the strategy used by previously described execution-based techniques used by frameworks like Tensorflow and PyTorch. Figure 2.17a shows a straight-line program with a *static* and *dynamic* parameter, and Figure 2.17b shows the generated partially evaluated program for the *static* input `x = 5`.

```
1 void program1 (int x, int y) {
2     int z = x + 1;
3     int w = y + (z - 3);
4     w = w + y;
5 }
```

(a) Program $\mathbb{P}_S$ with straight line code with *dynamic* parameters and intermediates in green

```
1 void program1 (int y) {
2     // static statements generate nothing
3     int w = y + 3;
4     w = w + y;
5 }
```

(b) Program $\mathbb{Q}_{SA}$ generated by evaluation of the program in Figure 2.17a for the *static* arguments `x = 5`

Figure 2.17: A simple staged program $\mathbb{P}$ with *static* and *dynamic* variables annotated and the generated program $\mathbb{Q}_{SA}$ for a specific input.

Since the variables `x` and `z` are annotated as *static*, they have concrete values during evaluation and generate no corresponding code in $\mathbb{Q}_{SA}$. The variables y and z, on the other hand, are placeholders and generate equivalent statements and expressions when operated on. When *static* values are

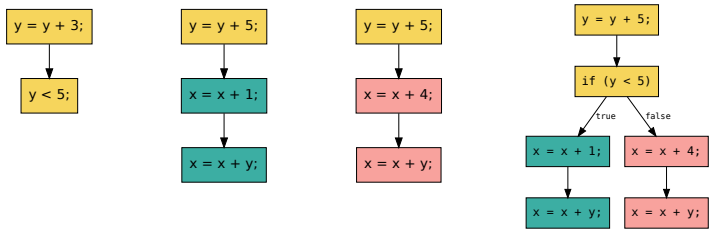
operated together with *dynamic* values, the concrete values of the *static* values are inserted as constants (eg, `z-3` is replaced with the value `3`).

Next, I will define the evaluation procedure for the simplest control-flow statement if-then-else, where the condition is an expression that depends on at least one *dynamic* value. The challenge is that since *dynamic* values don't have a concrete value during the evaluation of $\mathbb{P}_S$, the branch cannot be resolved, but executing both sides will violate the requirement for a semantically-correct execution order. We define the evaluation procedure for if-then-else as follows: for an input program p, when the evaluation reaches any if-then-else statement, the evaluation procedure halts the current evaluation E. Suppose the expression that if-then-else is branching on is e. It then creates two sub-evaluations E_T and E_F under the same evaluation procedure such that E_T continues the execution as if e had returned `true`, i.e., this execution starts executing the then branch. Similarly, E_F continues the execution as if e had returned `false`, i.e., it takes the else branch. The state of execution of E, including the program generated so far, is copied over to E_T and E_F. These sub-evaluations run to completion and produce the output programs q_{TA} and q_{FA} respectively. By construction, the two complete programs must have the same set of statements up to the point where the original program was halted. These statements are removed from both q_{TA} and q_{FA}, and one of the ideal pairs is added to a new program q_A. After these statements, an *if-then-else* node is added where the body of the then-branch is set to the remaining statements from q_{TA}, the body of the else-branch is set to the remaining statements from q_{FA}, and the condition is set to the program node extracted from e. q_A is finally returned as the result of evaluation of $eval_{static}$ on p. Notice that this procedure for handling if-then-else applies recursively. If during the evaluation of E_T or E_F, another if-then-else is encountered, the same procedure of starting two executions and combining the results is followed. We can observe that this evaluation procedure $eval_{static}$ not only runs multiple times per program, but also calls itself recursively. As an aside, this is where Re-Execution Based Multi-Staging (**REMS**) takes its name from.

We will analyze the number of executions later. However, first we observe that in all the recursive executions, the program always executes from the beginning, and for each if-then-else, it either takes the then or the else branch, but not both. Thus, all the executions are semantically correct execution orders. Second, we observe that for each if-then-else executed in the program, the generated program has an if-then-else. The then-branch of this if-then-else contains the rest of the program assuming the condition evaluates to true, and the else-branch contains the rest of the program assuming the condition evaluates to false. Although the generated program and the body of the if-then-else doesn't look like the original program, it is easy to see why the *dynamic* execution of this program would behave the same way when P is executed and thus would produce the same output b_A. Essentially, when the execution of the generated program reaches the generated if-then-else during the *dynamic* stage, the condition is evaluated with concrete values. If the condition evaluates to `true`, the execution continues to a program that was generated assuming this condition had evaluated to `true` during the `static` stage. Similarly for the `false` case. Thus, this is a valid partially evaluated program for the given $\mathbb{P}_S$ and arguments A.

Figure 2.18a shows an example program $\mathbb{P}_S$ with all parameters *dynamic* and an if-then-else condition based on one of the parameters. Figure 2.18b shows the extracted program from the execution till the point where the if-then-else is encountered. The expression e is shown as a standalone statement because in this execution, it is extracted before the if-then-else node is evaluated.

```
1  void program2 (int x, int y) {
2      y = y + 3;
3      if (y < 5) {
4          x = x + 1;
5      } else {
6          x = x + 4;
7      }
8      x = x + y;
9  }
```



(a) (b) (c) (d) (e)

Figure 2.18: a) A program p with a single if-then-else dependent on a *dynamic* value. b) Program extracted until the point the if-then-else is encountered. c) Complete extracted output program from E_T. d) Complete extracted output program from E_F. d) Merged program q_A.

Merging Trailing Code

While the program q_A generated by the evaluation procedure described above is correct, it has significant code duplication. Notice in Figure 2.18e, the last statement `x = x + y` is duplicated on both branches of the if-then-else. This problem is made worse if the statement itself is another if-then-else. All the statements appearing after the second condition will be duplicated 4 times, and so on. To avoid this exponential blowup in the generated code, we need to merge the statements that appear after the if-then-else. Instead of syntactically looking at where the if-then-else ends in the original program p, we merge statements if they are semantically equivalent. We modify the evaluation procedure in the following way: Once the evaluation procedure recursively extracts the programs from q_{TA} and q_{FA} from E_T and E_F respectively, we look for statements that are exactly equal in the two programs from the back. All these statements (which could include other if-then-else statements) are removed from the programs q_{TA} and q_{FA} and are added to q_A after the newly created if-then-else. This simple change collapses all the redundant copies of statements when applied recursively to all evaluations of $eval_{static}$ and produces a final q_A without any blowup in code. Figure 2.19 shows the generated code.

2.8.2 Controlling Complexity with Static Tags

The technique described in Section 2.8.1 shows how merging similar statements from the back in q_{TA} and q_{FA} post extraction avoids the exponential blowup in the size of the extracted program q_A. However, since the programs are entirely extracted before merging, the evaluation complexity of $eval_{static}$ still remains exponential in terms of the number of consecutive if-then-else statements. Very simply, every possible path created by taking the then or the else branches of different if-then-else statements is executed and merged into a single program, which would be naturally equal to 2^n where n is the number of consecutive if-then-else statements extracted.
This exponential complexity is due to the redundant work performed in the extraction of the code after the branches of a condition meet. This can be avoided by eager merging of executions E_T and E_F when the two branches of an if-then-else meet. However, we have seen in Section 2.6 that syntactic merging of branches can lead to inaccurate code being generated. There could be updates

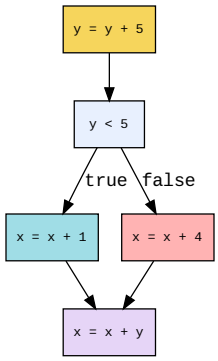


Figure 2.19: Optimized q_A for the program in Figure 2.18a with statements after the if-then-else merged into one to avoid blowup.

to *static* variables, which could produce different code even after merging.

To early merge the executions E_T and E_F, we need something that can detect early that two executions would run the exact same way and would produce the same output without having to run them to completion. We note that at any point, the rest of the execution of $eval_{static}$ depends on where the execution is currently at in the program P_S and the state of all variables that have concrete values. We need some way to encode/capture this runtime state of $eval_{static}$ so we can prove it equivalent to other points in the evaluation of $eval_{static}$ on the same program. We can then see that if two points during the evaluation of $eval_{static}$ evaluation the same encoding, their behavior from that point and then the output produced would be exactly the same.

To achieve this, I will introduce the notion of **static tags**. A static tag uniquely identifies any generated statement across all executions as part of an invocation of $eval_{static}$. We augment the behavior of $eval_{static}$ to create and attach a static tag to each program statement it generates. For each operation (binary, unary, or others), evaluated as part of $\mathbb{P}_S$, $eval_{static}$ records the program location of that operator in the representation of P_S. This program location can be anything, including line number and column number of the operator or the address of the instruction at runtime, as long as it uniquely identifies the operation in the program. If the language of $\mathbb{P}_S$ supports function calls, the program location includes the stack trace of the operation to uniquely identify each invocation of the functions from others. This program location becomes the first half of the static tag. The $eval_{static}$ further takes a snapshot (or a hash) of all the *static* variables live in the evaluation of $\mathbb{P}_S$ and appends them together. This concatenation of snapshots becomes the second half of the static tag. This newly created static tag is then attached to the generated program node for that operation to be inserted in $\mathbb{Q}_{SA}$.

Formally a static tag at an operation o, ST_o in the program $\mathbb{P}_S$ is defined as $ST_o = program\_location_o :: snapshot(v_{o0}) : snapshot(v_{o1}) : ...snapshot(v_{on})$ where $v_{oi} \in$ the set of all live variables annotated *static* at the time of the execution of o

We observe that a static tag ST captures the entire state of the evaluation of $\mathbb{P}_S$ under $eval_{static}$ since it contains where the evaluation is and the state of all *static* variables. The *dynamic* variables are only placeholders that do not have any concrete values. I argue that across all recursive invocations of $eval_{static}$ on a program $\mathbb{P}_S$ on a certain input A, if two operations produce the same static tag,

the rest of the evaluation from that point will exactly be the same, and the evaluation will produce the same program nodes. Since the evaluation of a program depends only on its state (program location and state of variables), if two evaluations have the same state, the program will execute deterministically in the same way, producing the same output. We can now use this property to eagerly merge executions.

We further augment the evaluation procedure $eval_{static}$ to maintain a map $\mathbb{MM}$ of all static tags that are created during the evaluation to the sequence fo program statements generated after that static tag was created ($\mathbb{MM} = ST \rightarrow [S_1, S_2, ...S_n]$ where S_i are statements generated as part of $\mathbb{Q}_{SA}$ after the operation with the static tag ST). This map is updated as soon as a sub-invocation of $eval_{static}$ concludes and a program q_A is ready to be returned. This map is also shared across all sub-invocations of $eval_{static}$ on $\mathbb{P}_S$ with the arguments A. Finally, during the evaluation of $eval_{static}$ when a new static tag is generated, the evaluation procedure checks if the new tag exists in the map $\mathbb{MM}$. If it does, the evaluation is immediately terminated and the rest of the program q_{ret} is filled in by inserting the sequence of statements stored in the map $\mathbb{MM}$ against the static tag.

Essentially, we apply memoization to the evaluation procedure $eval_{static}$ of a program $\mathbb{P}_S$, where we cache the rest of the generated program against static tags generated during the evaluation. We can now prove that with this change, the extraction complexity of $eval_{static}$ is reduced from exponential to polynomial in terms of the number of statements generated in the program $\mathbb{Q}_{SA}$.

Let us assume the number of statements produced as part of the final program $\mathbb{Q}_{SA}$ is m. First, we start by identifying the maximum number of possible invocations of $eval_{static}$ when the memoization is applied. We notice that two new invocations of $eval_{static}$ are created every time the execution reaches an if-then-else dependent on a *dynamic* variable. The condition of the if-then-else must be an operation on the *dynamic* variable, which must have a unique static tag. We claim that the condition of this if-then-else is converted into a program exactly once across all invocations. If there were two separate executions where the if-then-else with the same tag was executed, the memoization would cause the execution that runs later to terminate and copy over the remaining statements from the prior execution. This also guarantees that the number of invocations of $eval_{static}$ created is at most twice the number of unique static tags created. Since each statement generated as part of the final program $\mathbb{Q}_{SA}$ has exactly one unique tag, the number of invocations is at most twice the number of statements generated in $\mathbb{Q}_{SA}$ ($2m$).

Now, each invocation of $eval_{static}$ can run for at most m steps since each step produces a statement to be added to $\mathbb{Q}_{SA}$. Once again, if $m + 1$ statements are produced as part of any invocation, memoization would kick in, and the evaluation would be terminated.

Finally, the amount of work done to extract each statement is constant, but creating and comparing the static tag requires snapshotting and comparing all the live *static* variables. The maximum number of static variables live at any point (l) in the program is not related to m since an arbitrary number of *static* variables can be created during the evaluation of $\mathbb{P}_S$ if the language allows dynamic (heap) allocations in loops based on input parameters. However this $l < m$ for typical programs and thus the overall complexity can be approximated to $O(m^3)$ or more accurately $O(m^2 l)$ where m is the total number of statements produced in the output program $\mathbb{Q}_{SA}$ and l is the maximum number of *static* variables alive in during the evaluation of $\mathbb{P}_S$ with arguments A.

This technique of applying memoization to the evaluation procedure $eval_{static}$ greatly contributes to making the re-execution-based multi-staging not prohibitively expensive and thus practical for real-world applications.

2.8.3 Loop Extraction with Static Tags

In the previous sections, we have successfully augmented the evaluation procedure $eval_{static}$ to extract statements and if-then-else statements without violating correctness or without blowing up the output or the execution time. The next important control-flow operation to extract is loops. I will describe the extraction of the most canonical form of loops, the while-loop. The rest of the types of loops, like for-loops and the goto-loops, follow a similar procedure.
We change the evaluation procedure $eval_{static}$ to evaluate the while-loops where the condition is purely dependent on variables annotated *static* and program constants, just like regular loops where the condition and the body are evaluated repeatedly till the condition is true. This is because *static* variables and expressions written using them have concrete values during the evaluation of $eval_{static}$ on $\mathbb{P}_S$. Thus, the loop can be completely evaluated the required number of times.
For the loops where the condition depends on at least one *dynamic* variable, we have observed before that a syntax-based extraction methodology can lead to incorrect code being generated or even to undefined behavior being invoked. To get around this, we treat the condition of the while-loop just like an if-then-else.

We start with the claim that during a single evaluation of $eval_{static}$, suppose a static tag repeats, the execution will repeat infinitely and will produce the same code over and over as the code produced since the first time the static tag was generated. This is easy to prove, since the execution of $eval_{static}$ over $\mathbb{P}_S$ for arguments A depends entirely on the state captured in the static tag. If a static tag repeats, the execution state must have reached the same state as before, and the execution will continue in a loop. If the execution repeats in a loop, it will produce the same code.

With this observation, we change the evaluation procedure $eval_{static}$ in the following way: During an individual evaluation of $eval_{static}$, we keep a set $\mathbb{ML}$ of all static tags that have been generated for that execution. This set is not shared with any other executions. If a newly generated tag is already present in $\mathbb{ML}$, we terminate the execution and insert a `goto` statement in the generated program, jumping to the statement where the tag was generated for the first. Otherwise, we add the static tag to the set $\mathbb{ML}$ and continue execution. Whenever the execution of a program p reaches a while-loop where the condition is dependent on at least one *dynamic* variable, we treat it just like an if-then-else condition. Specifically, we replace the while-loop with an if-then-else with a goto at the end, going back to the condition and the same as the condition of the while loop. The else branch of the newly created if-then-else is set to empty. This if-then-else is then treated like a regular if-then-else, where two sub-evaluations E_T and E_F are created, which take the `true` and `false` respectively. These evaluations are then separately completed and merged to obtain a single program.

In the presence of any control-flow that goes back to statements seen before, such as a while loop or a goto, this evaluation procedure generates rudimentary goto-loops, with the gotos possibly inside a condition. We claim that this program is a correct extraction of $\mathbb{Q}_{SA}$. Since in the presence of repeated static tags during the execution of $eval_{static}$, the evaluation is bound to repeat and the same program is generated infinitely, it is equivalent to instead generating a program with the code just once, but a goto going back to where the repetition starts. When the hypothetical infinite program $\mathbb{Q}_{SA}$ is executed, it will repeat the statements between the two tags over and over, which is the same behavior when the program $\mathbb{Q}_{SA}$ with a goto is executed.

Thus, with the help of static tags and keeping track of static tags that repeat, back edges in the generated program can be identified, and rudimentary goto-loops can be created alongside if-then-

else and regular statements. With these control-flow constructs, any combination of control-flow can be handled, and multi-staging can be implemented in a semantically correct way using re-execution. Notice that the evaluation procedure described above isn't specific to a while-loop. The key idea is that the $eval_{static}$ explores all control flow paths by forking the execution at a condition based on a *dynamic* value and reconstructs control flow based on the observed generated programs. This means that the procedure can also transparently handle and extract other control-flow constructs into equivalent conditions, jumps, and loops, like for-loops, or even control-flow redirected by exception handling.

2.9 Analogy

In the above section, we saw how **REMS** uses multiple executions of the same program P_S that each take a different control-flow path and merges the results from each execution to produce a complete picture of the extracted program.

I would like to explain this with an analogy from maze solving. Suppose you are in the middle of the maze and trying to get out. You need to construct an entire map of the maze to find your way. If you had access to the birds-eye view of the whole maze, this would be trivial. However, if you can only look at the part of the maze around you with a small lamp in your hand, you can still construct a whole picture of the way. What you would do is keep exploring different paths in the maze in a systematic way, going down different paths, then backtracking when running into a dead-end, leaving markers so you don't trace the same path again and again, and so on. By doing this repeatedly and only with local information, you can construct a whole picture of the maze on a piece of paper and then solve it to find your way out.

REMS is similar in that it only accesses the local information by looking at the current operation it is executing. When it reaches a fork in the road (a condition based on *dynamic* variables), it explores all paths systematically and then merges all the information to construct a whole view of the program.

2.10 Interactions Between *static* and *dynamic* variables and Caveats

So far, we have discussed the prescribed methodology for extracting programs P_S containing *static* and *dynamic* variables. In this section, I will briefly cover how these two types of variables interact through various control flow constructs and what considerations should be taken.

2.10.1 *static* variable in a *dynamic* expression

. The simplest way a *static* and *dynamic* variables can interact is when a *static* variable appears in an expression containing at least one *dynamic* variable. Due to the existence of even on *dynamic* variable, the whole expression is *dynamic* and the *static* variable just becomes a constant in the generated expression. The value of the constant is equal to the concrete value of the *static* variable at that point. We call this process **simplification** since sub-expressions as part of bigger expressions are replaced with a constant in the generated code.

2.10.2 *dynamic* expression in the body of *static* control flow

When there is a control-flow like if-then-else of a loop that is purely dependent on a *static* variable, it is completely evaluated during $eval_{static}$. If this control flow contains any expressions of type *dynamic*, the control flow specializes in the creation of the statements containing these expressions. If the control flow is an if-then-else, the statements from the appropriate branch are selected based on whether the condition evaluates to `true` or `false`. If the control flow is a loop, the loop is completely evaluated, and the body of the loop containing the *dynamic* statement is duplicated/unrolled with each iteration specialized for that iteration. For example, if there is a loop with a condition inside based on the loop index, each iteration would be unrolled, and the branches would be completely resolved separately for each iteration. Notice that **REMS**'s early merging or back-edges doesn't kick in here since at least one *static* variable is changing, causing a different unique static tag to be generated each iteration. This kind of iteration is called **specialization** since the generated code is specialized based on the data in the *static* variables (like loop trip count). This technique is very powerful for building high-performance abstractions.

2.10.3 *static* updates in the the body of a `dynamic` control flow

Another interesting case is when a *static* variable is mutated inside an if-then-else or a loop that has its condition variable based only on *dynamic* variables. Normally, one would expect the condition or the loop to be generated as it is. But care has to be taken that when the execution exits the control flow, the *static* variable doesn't have the same value as the other branches. This means the generated code will not be merged into a single branch after the if-then-else or a loop back edge. If the control flow is an if-then-else, the two sides will continue executing separately, potentially leading to an exponential blowup for no reason. If it is a loop, and the *static* variable is constantly changing, it might lead to an infinite loop, which might not be the expected behavior. It is worth noting that any loop that has a *static* variable being updated inside automatically behaves like a *static* loop, leading to unrolling. This might sometimes be desirable, as we will see in Chapter 5, where if the *static* variable changes only once for the iterations, the first iteration is peeled out and the code converges after the second iteration. This might be desirable to specialize the first iteration in a different way.

2.10.4 Recursion containing only *dynamic* conditions

In **REMS**, care must be taken with recursive functions where the terminating condition is based purely on a *dynamic* value. Since *dynamic* variables have no concrete values, **REMS** will keep executing all branches and recurse infinitely. The static tag also won't be able to detect a loop or a recursion because the program location is changing with every stack frame being added. In such cases, to avoid infinite recursion, a guard variable of type *static* must be added to the function that can keep track of the recursion depth and return if it goes beyond a maximum value. This will avoid infinite recursions during the evaluation of $eval_{static}$. In future work, **REMS** might have to be modified to automatically detect recursion, like in loops, and insert a recursive call instead of infinitely recursing.

Chapter 3

BuildIt: Re-Execution based Multi-Staging in C++

In Chapter 2 we discussed how Re-Execution Based Multi-Staging (**REMS**) is the key to implementing multi-staging in imperative languages without introducing correctness issues due to side-effects on first stage variables. I also described a procedure for converting a single-stage program $\mathbb{P}$ into a staged version $\mathbb{P}_S$ given S and the implementation of a language-agnostic $eval_{static}$ that handles expressions, statements, and control-flow on *dynamic* variables using re-execution to explore all control-flow paths. In this Chapter, I will make the case for implementing **REMS** in a high-performance imperative language like C++ in a type-based manner. I will also discuss the specific challenges of implementing $eval_{static}$ in a lightweight manner without changes to the C++ compiler. Finally, I will describe BuildIt, a lightweight type-based multi-staging framework that is implemented using re-execution-based multi-staging. I will go over the user-API and the overall design of BuildIt

Figure 3.1: The BuildIt Framework Logo. BuildIt is available open source under the MIT license at https://buildit.so

3.1 Why Multi-Staging in C++?

Multi-Staging has several benefits over single-stage programs, including program reuse, zero-cost abstractions, and intermixing different programming languages, among others. However, one of the most important benefits of multi-staging that this thesis is centered around is obtaining high-performance through code generation while maximizing productivity. Most high-performance

libraries and implementations require low-level control over the execution, including manual allocations, order of execution and memory accesses, precise control over types and layouts, and infrastructure support for parallelization. While functional programming languages like Haskell, Scheme, and OCaml provide a higher-level abstraction that greatly improves productivity, they regularly lack low-level control (without massive implementation efforts and breaking the clean abstractions). At the same time, some imperative languages like Python, Java, and JavaScript do offer control over the order of execution of statements, but they abstract the details of object layouts and allocations from the user, which is crucial for extremely optimized high-performance code. Since these languages are either interpreted or run in a virtual machine, they incur extra overhead. As a result, languages like C, C++, Rust, or Julia, which are compiled either ahead of time or just in time, remain the de facto standard for high-performance implementations.

Naturally, these languages also have some notion of multi-staging or metaprogramming built into them to maximize the performance gains. C has basic support for generics through preprocessor macros, while C++ supports templates, constexpr, and consteval, which are not only more type-safe but also offer more features, making them Turing-complete. Rust offers metaprogramming and generics through the use of proc-macros, which can transform and generate programs at compile time by directly manipulating the program AST. Julia is a just-in-time compiled programming language targeting high-performance computational domains, offering specialization based on runtime values (and types) as a way of getting the benefits of multi-staging. However, all the existing multi-staging techniques are severely restrictive and do not offer user control and a seamless experience that is required to boost programming productivity while obtaining the best performance. Even though C++ templates are Turing complete, they require the user to write the first stage code in a completely different and restrictive syntax that makes moving computation between stages extremely difficult. At the same time, while C++ is powerful due to its rich features, including mutations, pointer indirections, heap allocations, support for external libraries, and object-oriented programming, templates do not offer any of these, further reducing the seamlessness of these frameworks. C++ consteval and constexpr do offer a similar syntax and some local mutations, but the overall set of features remains very restrictive. Rust's proc macros also require the developer to manually manipulate ASTs, which is akin to writing compiler passes, which makes the whole process tedious. At the same time, proc macros run at compile time and cannot accept parameters or call external libraries (written as general Rust code). Julia offers minimal user control over what is specialized at runtime and leaves the choice to the compiler, making it difficult to implement low-level optimizations. All these languages and frameworks also implement multi-staging in an implicit way, making it extremely difficult to debug generated code for both correctness and performance.

However, among all the high-performance languages, C++ is the most feature-rich and one of the oldest languages. Naturally, a lot of high-performance libraries and applications have been written in C++, including boost [1], BLAS [179], Eigen [81], libtorch [139], game engines among many others. At the same time, most high-performance developers also have expertise in writing and optimizing C++. C++ also has support for generics and polymorphism through operator and function overloading, which, as we will discuss, is a key feature for implementing multi-staging in a seamless manner. Finally, many other high-performance languages like CUDA and HLS also derive from C++, allowing us to extend multi-staging support to these languages. As a result, C++ is the ideal choice for implementing a multi-stage framework to maximize the applicability to high-performance domains.

3.1.1 Challenges for multi-staging in C++

C++ is a very suitable language to implement multi-staging since it is an imperative language that is a great fit for high-performance computing. However, C++ comes with its own unique challenges. Firstly, C++ is a statically typed compiled language, meaning no information about the program is available at runtime. Specifically, C++ lacks support for runtime reflection, which can be a way to aid the implementation of **REMS** to look for the start and end of control flow on *dynamic* variable. Other languages, like Python, that can inspect the AST at runtime, can use this to modify $eval_{static}$.

Secondly, even though C++ has great support for polymorphism through operator overloading and generics, it doesn't have support for overloading control flow like if-then-else or loops. Due to these reasons, multi-staging in C++ has been limited in the past, either simply not allowing control-flow or using a syntactic sugar like a function that accepts lambdas for control flow [10], which we have seen can lead to the side-effects leak problem. I will explain how we sidestep all these issues.

3.2 The BuildIt Multi-Staging Framework

In this section, I will introduce and describe BuildIt a multi-stage execution framework embedded in C++. I will describe the design choices, including the specific flavors of multi-staging, the procedure to convert a single-stage program to a multi-stage program, and the actual implementation of the evaluation procedure of BuildIt.

3.2.1 A Lightweight Approach to Multi-Staging

C++ is one of the earliest high-performance languages created in 1979. Since then, it has come a long way, including many new features and language versions being added (C++98, C++11, C++14, C++17, C++23). Naturally, the language design is extremely complicated, as evident by the fact that the C++11 language standard (N3337 [66]) is about 1,353 pages long. As a result, C++ compilers are extremely complicated pieces of software with multiple layers and stages of frontend, midend, optimizations, and code generation. The clang+LLVM compiler is over 35.6 million lines of C++ code.

As a result, changing the C++ language and the compiler to add support for multi-staging is a near-impossible task. Furthermore, most high-performance C++ projects are massive implementations, spanning across multiple repositories that rely on compiler vendor extensions to language features and runtime libraries. Adding support for a custom compiler that supports multi-staging would require massive changes to these repositories to migrate the source code to work with the new compiler.

C++ developers, however, are extremely comfortable with using pre-compiled and header-only libraries since most projects already depend on optimized libraries. Most C++ build systems (and build system generators) like make, ninja, and CMake make it extremely easy to add dependencies on libraries. Finally, existing C++ features like operator overloading, exceptions, closures, and templates are surprisingly enough to implement a feature-rich multi-staging framework as I will describe in the next chapter. As a result, C++ was chosen to implement the BuildIt system as a pure library that can be used by including a few headers and by linking against a runtime library.

We call this approach a lightweight approach as opposed to a heavyweight approach that requires language and compiler changes. Formally, this means that we define $eval_{static}$ to be exactly the same $eval_{C++}$ where $eval_{C++}$ is the language specification and execution environment of C++. Specifically, BuildIt is implemented to be compatible with C++11, for maximum backward compatibility and for supporting old code bases. This design decision makes it extremely easy to integrate and apply our system to existing high-performance code bases.

3.2.2 A Type-Based Multi-Staging Approach

In Chapter 2, we discussed different flavors of multi-staging and their tradeoffs in features and seamlessness. Type-Based multi-staging where the program representation of the staged program ($\mathbb{P}_S$) differs from an unstaged program ($\mathbb{P}$) only in the declared types (or type annotations) of the variables and intermediate expressions is one of the most seamless flavors of multi-staging. Not only does it make it easy to port existing single-stage programs into multi-stage equivalents, but it is also extremely malleable and allows changing the binding times of variables and expressions (which stage they are evaluated in) extremely easily by requiring only changes to the types. Furthermore, C++ makes it extremely easy to write generic code over types using templates. These features can further be combined with the staging types to offer even more flexibility. As a result, I chose to implement BuildIt as a type-based multi-staging library. I will show an example in the later sections that combines C++ templates with multi-staging to make the experience even more seamless.

3.2.3 Explicit Multi-Staging

The existing metaprogramming capabilities present in C++, including templates and constexpr/consteval, are offered as implicit techniques where the generated program is an intermediate inside the compiler that the programmer cannot view or modify. While this makes the compilation pipelines easy, it makes it extremely hard to debug the generated code for both correctness and performance. To offer maximum flexibility to the user, we implement BuildIt as an explicit multi-staging framework where the output of the first stage is explicit source code printed as a string, which can be viewed and modified before compiling and executing again. Moreover, this approach also allows us to mix C++ with adjacent languages like C, CUDA, and HLS by very slightly tweaking the generated code. Finally, this also allows us to leverage existing C++ extensions and frontend constructs like OpenMP and Cilk that allow us to add parallelization by simply generating some annotations.

3.3 The BuildIt Programming Model and API

In this section, I will explain the programming API for BuildIt, the types and function calls it introduces, and the constraints that need to be followed for writing well-behaved BuildIt programs. I will also discuss the API functions for some of the internal parts of BuildIt, which might be relevant to advanced users extending BuildIt.

3.3.1 BuildIt namespaces

Since BuildIt is just a library in C++, it can be used by directly including the headers that provide different types and functions. The API for BuildIt is divided into two namespaces: the `block` namespace and the `builder` namespace, which are respectively available under the `blocks/*` and `builder/*` set of headers. The `block` namespace has types and functions to represent the extracted code as part of $\mathbb{Q}_{SA}$. BuildIt uses an Abstract Syntax Tree (AST) representation containing variables, binary and unary expressions, assignments, if-then-else statements, and while and for-loops enclosed inside functions. Since the framework is embedded in C++ and generates C or C++, the ASTs are also typed, meaning each generated variable and function has types associated with it. The `block` namespace also defines the base classes for visitors and replacers to analyze and manipulate the ASTs before generating code. Finally, the `block` namespace contains a variety of passes for performing correctness and readability transformations to the AST, and finally the `c_code_generator` visitor for printing C or C++ source code for $\mathbb{Q}_{SA}$. The `block` AST is the intermediate representation BuildIt uses to represent the code during extraction and for post-processing.
The `builder` namespace, on the other hand, contains all the types and functions that assist with the actual extraction of the generated program and the actual API exposed to the user. This mainly includes the two type templates `dyn_var<T>` and `static_var<T>`, which are described next, and the `builder_context` class that allows the user to supply options and also serves as an entry point to BuildIt's extraction process.

3.3.2 The `dyn_var<T>` and `static_var<T>` types

As explained before, BuildIt takes a type-based approach to multi-staging. This means the declared type of the variables and expressions decides what stage they are evaluated in, i.e., whether a variable is *static* or *dynamic*. The procedure to convert a single-stage program into a multi-stage program involves applying the $\mathbb{D}$ and $\mathbb{S}$ type-level functions to the declared types of all variables and function arguments and return types based on what needs to be specialized. The first step in describing $eval_{static}$ is to describe these functions. We introduce two new type templates `dyn_var<T>` and `static_var<T>` which correspond to *dynamic* and *static* variable types. For BuildIt, the type functions, $\mathbb{D}$ and $\mathbb{S}$ are declared as follows:

$$\mathbb{D}(T) = \texttt{dyn\_var<T>}$$

$$\mathbb{S}(T) = \begin{cases} \texttt{T} \text{ or } \texttt{static\_var<T>} & \text{, if T is } \texttt{const} \text{ or the variable is never mutated} \\ \texttt{static\_var<T>} & \text{, otherwise} \end{cases}$$

This means, the variables and expressions that are to be executed in the second stage are simply all declared as `dyn_var<T>` where T is the original type of the variable. For the variables that are to be executed in the first stage (during $eval_{static}$), the variables need to be declared as `static_var<T>`. However, if the variable is declared `const` or is not declared `const` but does not change during the execution of the sub-program being staged, the `static_var<T>` type wrapper can be omitted and the regular C++ type can be used. There are a few restrictions on the types that can be wrapped in `dyn_var<T>` and `static_var<T>`, and handling custom types, which would be explained

in Chapter 4. Since BuildIt uses type-based multi-staging, apart from the no other part of the function implementation needs to be changed. To summarize, the types mentioned above enforce the following constraints on valid BuildIt programs -

- All *dynamic* parameters and values that are directly assigned from one or more *dynamic* parameters or other *dynamic* values must be declared using the `dyn_var<T>` type.
- All the rest of the values in the program that mutate their value during the execution at least once must be declared using the `static_var<T>` or `dyn_var<T>` type.
- All the remaining values in the program that do not mutate once set can be declared as `static_var<T>`, `dyn_var<T>`, or using regular C++ values.

With these constraints in mind, I will now explain how a program ($\mathbb{P}_S$) in BuildIt is written. Let us look at converting a simple **power** function that accepts the base and exponent and returns $base^{exponent}$ using repeated squaring. Figure 3.2 shows the implementation of the power function as a single-stage function in C++ ($\mathbb{P}$). The implementation of this function accepts two integers **base** and **exp**. The main body of the function is a while loop that runs till the exponent is greater than 0 and halves the exponent every step while squaring the base (initialized to the base). Furthermore, at each step, if the exponent is odd, the current base is multiplied into the result. This implementation computes the exponent in $log(exp)$ steps.

```
1  int power (int base, int exp) {
2      int res = 1, x = base;
3      while (exp > 0) {
4          if (exp % 2 == 1)
5              res = res * x;
6          x = x * x;
7          exp = exp / 2;
8      }
9      return res;
10 }
```

Figure 3.2: Implementation of the power function as a single-stage C++ program using repeated squaring.

Figure 3.3a shows the function modified for multiple stage execution where the base is *dynamic* and the exponent is *static*. Specifically, for the function **power** if $x_0 = base$ and $x_1 = exp$, $S = \{1\}$ and consequently $D = \{0\}$. Firstly, since these types are part of the infrastructure to construct the generated code, these types are provided under the `builder` namespace under the headers `builder/dyn_var.h` and `builder/static_var.h`. These headers are included, and the types are brought into the current namespace. Next, the return type of the **power** function is changed from `int` to be `dyn_var<int>` as required by the modification procedure for type-based staging. Since the parameter **base** is *dynamic*, its type is changed from `int` to `dyn_var<int>`, and since the parameter **exp** is *static*, its type is changed from `int` to `static_var<int>`. Furthermore, since the **exp** variable actually gets mutated during the execution, the wrapping with `static_var<T>` cannot be skipped. Finally, since

```
#include <builder/dyn_var.h>
#include <builder/static_var.h>
using builder::dyn_var;
using builder::static_var;

dyn_var<int> power (dyn_var<int> base,
        static_var<int> exp) {
    dyn_var<int> res = 1, x = base;
    while (exp > 0) {
        if (exp % 2 == 1)
            res = res * x;
        x = x * x;
        exp = exp / 2;
    }
    return res;
}
```

(a) Implementation of the power function as a two-stage program where the base is *dynamic* and exp is *static*.

```
#include <builder/dyn_var.h>
#include <builder/static_var.h>
using builder::dyn_var;
using builder::static_var;

dyn_var<int> power (static_var<int> base,
        dyn_var<int> exp) {
    dyn_var<int> res = 1, x = base;
    while (exp > 0) {
        if (exp % 2 == 1)
            res = res * x;
        x = x * x;
        exp = exp / 2;
    }
    return res;
}
```

(b) Implementation of the power function as a two-stage program where the base is *static* and exp is *dynamic*.

Figure 3.3: To BuildIt-ified versions of the **power** function specialized for **exponent** and**base** respectively by adding BuildIt types

the two intermediate variables, **res** and **x**, are dependent on a variable declared as **dynamic**, they are required to have their types changed from **int** to **dyn_var<int>**. The rest of the function and its implementation stay exactly the same as in the original program.

To demonstrate the malleability of BuildIt's API, Figure 3.3b shows another multi-staged version of the power function where the base is *static* and the exponent is *dynamic*. For this version of **power**, $S = \{0\}$ and $D = \{1\}$. Notice that the only difference between this version and one in Figure 3.3a is that the types for **exp** and **base** have been swapped around. The rest of the code is exactly the same. Thus highlighting how easy it is to move execution between stages as optimization goals change.

In fact, as mentioned before, C++ templates can be combined to make a single version that implements both based on how the templates are instantiated to further improve the malleability. Figure 3.4 shows the templated version with template parameters **BT** and **ET** used to declare the two arguments. The last two lines show two instantiations of the template; each behaves like the versions shown before.

3.3.3 Invoking BuildIt

After changing the declared types of variables, a BuildIt program needs to be told where the program to be staged begins. An evaluation of $eval_{static}$ is managed completely under a **builder::builder_context** An object which tracks the running state of $eval_{static}$, including invocation of recursive calls. Figure 3.5 shows the **main** function that actually creates a **builder::builder_context** object and calls the **extract_function_ast** function. This function accepts as its first parameter the entry point to the function where the program being staged begins, in our case, the **power** function. Notice that this is just the entry point to the program. This function can internally call other functions just like any

```
...
// Template parameters for the types of the two parameters
// The return type is still dyn_var<int> since the return
// value will always be in the dynamic stage.
template <typename BT, typename ET>
dyn_var<int> power (BT base, ET exp) {
    dyn_var<int> res = 1, x = base;
    while (exp > 0) {
        if (exp % 2 == 1)
            res = res * x;
        x = x * x;
        exp = exp / 2;
    }
    return res;
}
...
// Two instantiations of the power template
auto &power1 = power<dyn_var<int>, static_var<int>>;
auto &power2 = power<static_var<int>, dyn_var<int>>;
```

Figure 3.4: A BuildIt-ified version of the **power** function that also combines C++ templates to further improve malleability.

regular C++ program, and all the called functions would automatically become part of the generated program. The second parameter is the name to be assigned to the generated function. The rest of the parameters are the set of *static* arguments $a_i : i \in S$ for invoking $eval_{static}$ in increasing order of i. Notice that the *dynamic* arguments are skipped since they will be accepted by the generated function. In our case, since the function only accepts one *static* parameter **exp**, we pass the exponent, which is a command-line argument. Notice, unlike C++ templates, the value to specialize on doesn't need to be a literal constant and can be any ordinary program value. This function returns an opaque **block** AST object which can then be passed over to the **c_code_generator** for printing the source code for the extracted $\mathbb{Q}_{SA}$ to the standard out (or any file).

Figure 3.6a shows the result of the output from the staged program in Figure 3.3a when the exponent is supplied as 13. Right away, we notice that the while-loop iterating over the exponent is completely gone, and we see a sequence of multiply statements. This is because the loop depends entirely on a *static* value and can be completely unrolled. Furthermore, each iteration of the loop is specialized for the value of the **exponent** at that iteration. Notice that the if condition inside the iteration is also evaluated away, and the body of the if condition is added when the exponent is added. The division on the exponent is also removed. This kind of specialization allows us to eliminate branches and even some operations that would add runtime overhead. The generated code only has operations that depend on the *dynamic* values and thus is much easier to optimize for the C++ compiler.

Figure 3.6b, on the other hand, shows the output from the staged program in Figure 3.3b when the base is supplied as 13. In this case, since the control-flow, like while loop and if-then-else, depend on the *dynamic* input, they are generated as it is, and the *static* parameter **base** just becomes a constant in the code. This version doesn't optimize the code much but demonstrates how BuildIt

```
#include <iostream>
#include "builder/dyn_var.h"
#include "builder/static_var.h"
#include "builder/builder_context.h"
#include "blocks/c_code_generator.h"
using builder::dyn_var;
using builder::static_var;

dyn_var<int> power(dyn_var<int> base, static_var<int> exp);

int main(int argc, char* argv[]) {
    // Check if the program has the exponent argument supplied
    if (argc < 2) {
        std::cerr << "Please supply the exponent as a parameter" << std::endl;
        return -1;
    }

    // Create a new builder_context object and extract the AST for the power function
    // with the supplied exponent
    builder::builder_context ctx;
    auto ast = ctx.extract_function_ast(power, "power_specialized", atoi(argv[1]));

    // Pass the extracted AST to the c_code_generator for printing the source code
    // of the extracted program to std::cout
    block::c_code_generator::generate_code(ast, std::cout, 0);

    return 0;
}
```

Figure 3.5: The main function to invoke $eval_{static}$ and extra the second stage code for the **power** function

can extract very different code with small changes to specialization choices.

3.3.4 Naming Variables

Notice in the above examples that the generated code has auto-generated names like **arg0**, **var0**, etc. Since BuildIt is a purely library-based approach and C++ doesn't have reflection support for code, it has no means to access the names of the variables. This is usually okay for most applications since the generated code is only meant to be compiled and run again. However, there are two options if the user needs non-generated names for variables, if they want to improve readability.
The easiest way to preserve the names of the variable is to compile the BuildIt library itself with the **RECOVER_VAR_NAMES=1** build system flag. This forces BuildIt to be compiled with debug symbols enabled. The program, then written with BuildIt, also needs to be compiled with **-g** (or equivalent compiler flag for enabling debug support). With this option enabled, BuildIt can now look at the names of the variables stored as debug symbols in the binary and produce them for the user. Figure 3.7 shows the generated code for the output from Figure 3.3a when **RECOVER_VAR_NAMES=1** is enabled. Notice that the generated variables still have a suffix **_2**, **_1** inserted. This is to disambiguate

```
int power_specialized (int arg0) {
  int var0 = arg0;
  int var1 = 1;
  int var2 = var0;
  var1 = var1 * var2;
  var2 = var2 * var2;
  var2 = var2 * var2;
  var1 = var1 * var2;
  var2 = var2 * var2;
  int var3 = var1 * var2;
  return var3;
}
```

(a) Generated code when the **power** function in Figure 3.3a is run with the driver in Figure 3.5 with command line argument *13*.

```
int power_specialized (int arg1) {
  int var0 = arg1;
  int var1 = 1;
  int var2 = 13;
  while (var0 > 1) {
    if ((var0 % 2) == 1) {
      var1 = var1 * var2;
    }
    var2 = var2 * var2;
    var0 = var0 / 2;
  }
  int var3 = var1 * var2;
  return var3;
}
```

(b) Generated code when the **power** function in Figure 3.3a is run with the driver in Figure 3.5 with command line argument *13*.

Figure 3.6: Outputs from the two **power** functions written with BuildIt types

variables that have been duplicated, for example, due to loop unrolling.

```
int power_specialized (int arg0) {
  int res_1 = 1;
  int x_2 = arg0;
  res_1 = res_1 * arg0;
  x_2 = x_2 * x_2;
  x_2 = x_2 * x_2;
  res_1 = res_1 * x_2;
  x_2 = x_2 * x_2;
  return res_1 * x_2;
}
```

Figure 3.7: Generated program for the input program in Figure 3.3a when compiled with **RECOVER_VAR_NAMES=1** config option enabled.

However, sometimes the user might need exact names to be controlled. They might want to assign a different name from the name in the source code, or might need a precise name for a variable or a function to interact with external libraries. To support this, BuildIt has the **builder::with_name** copy constructor helper. This constructor accepts a string as a name **dyn_var<T>** variables copy-initialized from objects of these types, and gets the exact name. Figure 3.8 shows an example program with **with_name** used at the local and global scope. Notice that **dyn_var<T>** types can also be wrapped around function types to create *dynamic* function objects that just insert a **function_call_expr** whenever called. This is handy for calling external library functions that the developer doesn't have source access to or just doesn't need to BuildIt-ify. Notice that using **with_name** doesn't require **RECOVER_VAR_NAMES=1** or debug symbols to be enabled. The name assigned to the variable also doesn't

have to be the same as the name of the variable (**errno** in the example). Figure 3.9 shows the output when this program is run. Notice the precisely generated names for the functions, global variables, and local variables. This part of the API deviates from the type-based approach, but is, in most cases, not necessary for correctness and only serves to improve readability.

```
using builder::with_name;

// Wrap these global objects in a separate namespace so they don't conflict
// with the actual errno and atoi
namespace runtime {

// Global variable to access the errno variable in the generated code
dyn_var<int> myerrno = with_name("errno");

// Global variable to call an external function in the generated code
dyn_var<int (const char*)> atoi = with_name("atoi");
}

dyn_var<int> parse (dyn_var<const char*> str) {
    // Local variable with name assigned
    // The optional second parameter is set to true
    // tells buildit not to skip a declaration
    dyn_var<int> val = with_name("value", true);

    // Call to external function
    val = runtime::atoi(str);

    // Access to global variable with exact name
    if (runtime::myerrno != 0)
        return -1;

    return val;
}
```

Figure 3.8: Example program using the **with_name** copy constructor helper to assign specific names to variables.

```
int parse_specialized (char const* arg0) {
  int value;
  value = atoi(arg0);
  if (errno != 0) {
    return -1;
  }
  return value;
}
```

Figure 3.9: Output from the program in Figure 3.8.

3.3.5 Wrapping `dyn_var` Around User-Defined Types

The above examples show programs with only primitive types used. However, BuildIt also allows wrapping the `dyn_var` types around user-defined types like structs. BuildIt supports two kinds of user-defined types. Opaque types and non-opaque types. Opaque types are similar to the opaque types in C and C++, in that the program just cares about the name of the type. The members of the type are not relevant. An opaque type to be generated in BuildIt can be simply created using the `builder::named<T, Args...>` template. This template accepts a global string as a name and creates a type that can be wrapped inside `dyn_var`. Whenever the variables of this type are created, the exact name is used in place of the type. `builder::name` also accepts optional extra template arguments that are forwarded to the generated type. If the members of the user-defined types are required in the code, i.e., the type is not opaque, BuildIt requires the user to insert a `struct` definition of the type and convert all members to `dyn_var<T>`. The whole `struct` can then be wrapped inside `dyn_var<T>`. Just like local and global variables can optionally be given names using `with_name` if the auto-generated member names are not preferred. The whole type can also be given a name by adding a `static const char*` member to the type called `type_name` set to the desired name. Figure 3.10 shows an example program with both opaque and non-opaque user-defined types, and Figure 3.11 shows the generated output for this program. BuildIt also provides the utility `block::c_code_generator::generate_struct_decl<T>()` function to generate the definition of a user-defined struct in the generated code.

3.3.6 From Two Stages to N stages

So far, we have seen the programming API for writing code in two stages using `dyn_var<T>` for the second stage and `static_var<T>` for the first stage. Generalizing this to multiple stages is simply achieved by nesting the `dyn_var<T>` and `static_var<T>` types inside `dyn_var<>`. So we can declare variables with types `dyn_var<dyn_var<T>>` which, when evaluated, produce a program with `dyn_var<T>`. Similarly, when `dyn_var<static_var<T>>` is evaluated, it produces a program with `static_var<T>`. This can then again be evaluated like a normal BuildIt 2-stage program, thus effectively making a 3-stage program. These types can be arbitrarily nested as many times as the number of stages required. Notice that `static_var<T>` cannot be wrapped around BuildIt types since it is the same as just writing them directly as regular `static_var<T>` or `dyn_var<T>`.

3.3.7 The `block` Namespace

Before I get into how exactly the BuildIt implements the **REMS** based multi-staging, I will explain the internal representation that BuildIt uses to represent the program while extraction. Even though these data structures aren't part of the API that most users need, it is needed by users writing custom passes to extend BuildIt's features. Understanding these data structures will also help us understand the extraction procedure better. As explained above, the data structures used to represent the program are implemented under the `block` namespace as a set of AST classes. The root node of all AST nodes that represent operations, statements, variables, constants, and even types is the `block` class. Figure 3.12 shows the inheritance diagram for these classes. Under the `block` class, the top-level classes are `expr`, `stmt`, `label`, `var`, `type` and `trans_unit`. These are used to represent expressions, statements, labels, variables, types of variables, and entire translation units together. Under the `expr`

```
// Definition of the opaque type FILE
// without template args
static const char file_t_name[] = "FILE";
using MY_FILE = builder::name<file_t_name>;

dyn_var<FILE* (const char*, const char*)> my_fopen = with_name("fopen");

// Definition of the opaque named type template std::vector
static const char my_vector_t_name[] = "std::vector";
template <typename T>
using my_vector = builder::name<my_vector_t_name, T>;

// Definition of a non-opaque type
struct cylinder {
    // Assign a name to the struct by adding a member type_name
    static constexpr const char* type_name = "struct cylinder";

    dyn_var<float> rad; // Member with auto-generated name
    // Assign names to the members with with_name
    dyn_var<float> height = with_name("height");
};

dyn_var<float> volume(dyn_var<const struct cylinder&> c){
    dyn_var<float> v =
    3.14 * (c.rad * c.rad) * c.height;
    return v;
}

dyn_var<float> foo (void) {
    // Use a pointer to the opaque type FILE
    dyn_var<MY_FILE*> f = my_fopen("myfile.txt", "r");

    // Use the opaque template type std::vector<T>
    dyn_var<my_vector<int>> x = {1, 2, 3};

    dyn_var<struct cylinder> c;
    c.rad = x[0];
    c.height = x[1];

    return volume(c);
}
...
// As part of main, ask BuildIt to generate the definition for the struct
block::c_code_generator::generate_struct_decl<dyn_var<struct cylinder>>(std::cout, 0);
```

Figure 3.10: Program showing various opaque and non-opaque user-defined types.

```
struct cylinder {
  float mem0;
  float height;
};
float foo_specialized (void) {
  FILE* var0 = fopen("myfile.txt", "r");
  std::vector<int> var1 = {1, 2, 3};
  struct cylinder var2;
  var2.mem0 = var1[0];
  var2.height = var1[1];
  return (3.14 * (var2.mem0 * var2.mem0)) * var2.height;
}
```

Figure 3.11: Output from the program in Figure 3.10 showing the code with user-defined types generated.

class, there are various types of expressions, including **sq_bkt_expression** (square bracket expression), **functional_call_expr**, and others. The two primary types of expressions are **unary_expr** and **binary_expr** (shown in Figure 3.13) which have expressions like **not_expr** and **plus_expr** respectively. Most operations in C++ fall under one of these categories. Notice that **assign_expr** is also part of the **binary_expr** and is not a statement since assignments are also expressions in C++ and can appear as part of other expressions. The next major category under **block** is the **stmt** which has classes like **if_stmt**, **while_stmt**, **return_stmt** and even entire **function_decl** statements. These are all extracted by the control-flow extraction mechanism of BuildIt. One important class to look at is the **expr_stmt** that just holds an expression that appears as a standalone expression. These appear when expressions aren't used as part of other expressions and are cast to void. The last major category is the **type**, which has under it classes to represent different types that can be attached to variables like **scalar_type**, **pointer_type**, **array_type**, etc. Just like expressions and statements, these types can be nested to allow for creating complex types. Finally, the **block** namespace has AST nodes for representing constants, gotos, jump targets, etc. These nodes have pointers to other AST nodes stored inside them. For example, each **binary_expr** node has two **expr*** (pointer to expressions) nodes to represent the AST for the LHS and the RHS expressions. Similarly, the **stmt_block** AST node has a vector of statements nested inside it.

Besides the AST representation itself, the **block** namespace has two main utility classes - the **block_visitor** and the **block_replacer**. These classes implement the visitor pattern [136] to quickly iterate and modify AST nodes without having to write a large amount of code. These classes define a virtual function **visit** for each of the above AST class node that have the default behavior of visiting the children recursively. User-defined visitors can inherit from these base classes and override the functionality to perform analysis and transformations. Remember that these APIs are only meant for advanced users who have familiarity with compiler techniques. Regular users and high-performance DSL designers never need to touch the **block** namespace.

That wraps up the description of BuildIt's programming API for both regular users and advanced users looking to extend BuildIt's functionality with custom passes. In the next Chapter, I will explain how **REMS** is implemented in BuildIt to extract the code.

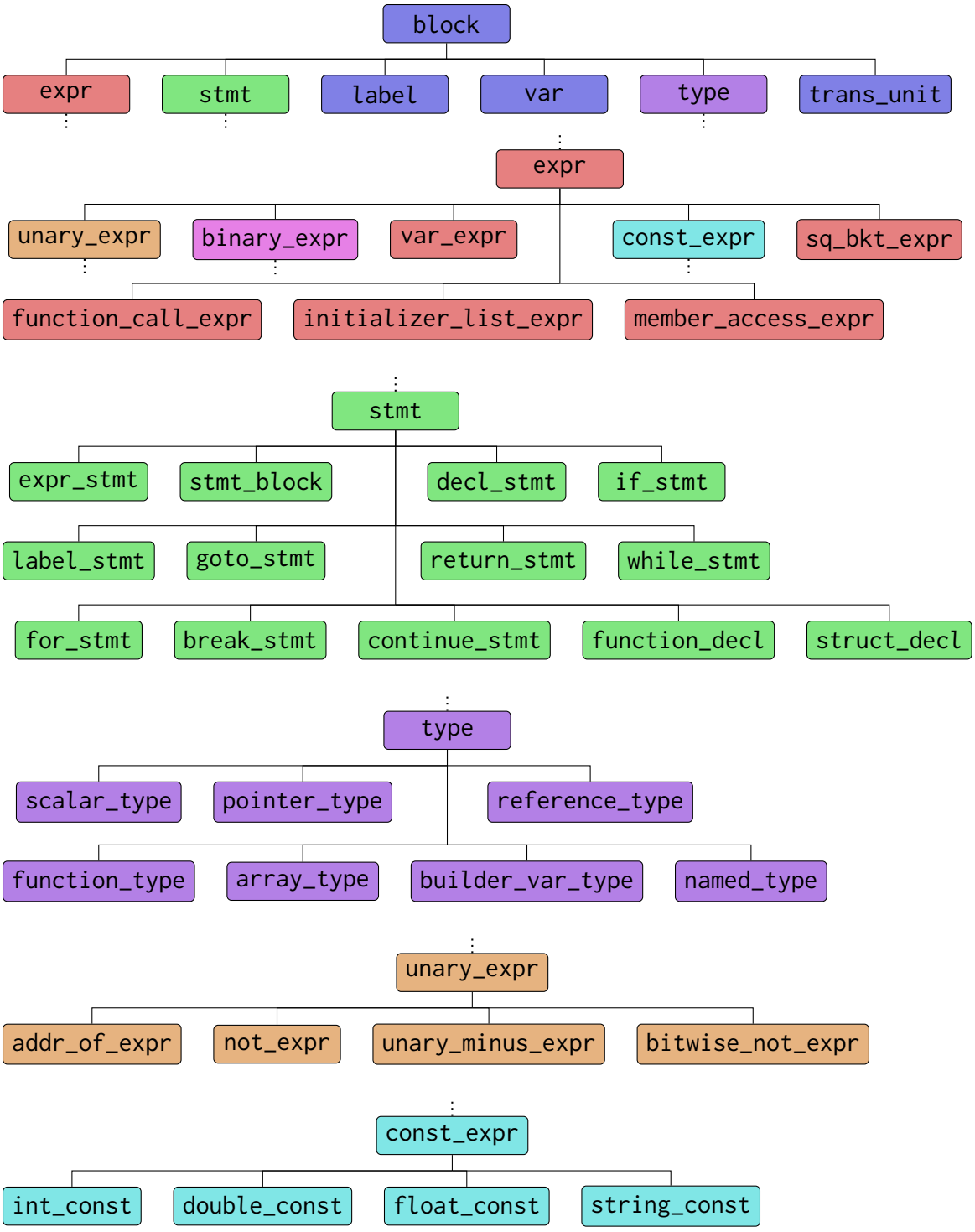


Figure 3.12: Inheritance diagram for the abstract classes **block**, **expr**, **stmt**, **type**, **unary_expr**, **const_expr** and their derived classes from the **block** namespace

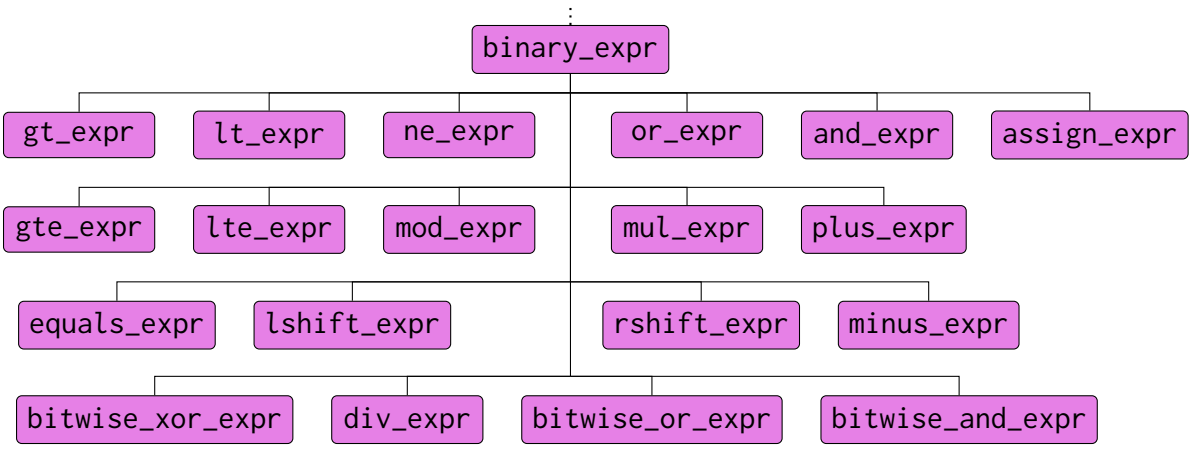


Figure 3.13: Inheritance diagram for the abstract class **binary_expr** and its derived classes from the **block** namespace

Chapter 4

Implementing REMS in BuildIt

In the previous chapter, I introduced BuildIt, the re-execution-based multi-staging framework in C++ that uses a type-based and lightweight interface. I also discussed the entire programming model of BuildIt and uncovered some of the internal representations. In this Chapter, I will deep dive into how exactly **REMS** is implemented without modifying the compiler in a language like C++ that also doesn't support reflection of types or code. Essentially, I will walk through each of the steps of implementing $eval_{static}$ as explained in Chapter 2 and explain how BuildIt implements it. I will start with how the `static_var<T>` type is defined in a way that the variables have concrete values during execution and how `dyn_var<T>` and others are implemented to extract semantically correct code for $\mathbb{Q}_{SA}$.

4.1 Handling *static* variables and expressions

As proposed in Chapter 2, the very first step in defining $eval_{static}$ is to handle the behavior of operations and control-flow on variables and expressions annotated *static*. Since we are implementing BuildIt in a lightweight library way, the implementation all needs to be done without modifying the compiler. Recall that in **REMS**, variables and expressions annotated *static* have a concrete value during the execution of $eval_{static}$ and are operated as regular host language types. The simplest way to implement this behavior is by declaring the type template `static_var<T>` as a wrapped type to `T` that automatically converts back and forth to `T`. The implementation of this type template is shown in Figure 4.1.

Notice that the implementation of the `static_var` type template is extremely simple. We define it as a class with a single member `val` which contains the concrete value stored in this variable. Further, we define `operator T` (with const) and `operator=(const T&)` that allows automatic conversion of `static_var` to and from `T`. This also handles all operations on `static_var`. Since C++ supports implicit conversion, whenever a `static_var` is used in an operation like addition, member access, or others, it automatically converts to `T` and implements the behavior.
We also define the constructors and destructors for the `static_var<T>` types. I will describe the implementation of these later when I explain how static tags are implemented in BuildIt.

```
namespace builder {
template <typename T>
class static_var {
    T val; // The wrapped value stored as a member

public:
    static_var() {...}
    static_var(const T& v): val(v) {...}
    static_var(const static_var<T>& v): val(v.val) {...} // Basic and copy constructors

    operator T& () {
        return val;
    }
    operator const T& const () { // operators for automatic conversions to T
        return val;
    }
    const T& operator= (const T& t) { // operators for assignment to T
        val = t;
        return val;
    }

    ~static_var() {...}
};
}
```

Figure 4.1: Definition of the `static_var` type template as a wrapper type to `T` with automatic conversion operators

4.2 Handling *dynamic* variables and expressions

Most of the complexity of **REMS** is in how $eval_{static}$ handles the *dynamic* variables and expressions. Recollect that variables of these types do not have a concrete value and are just placeholders that generate code when operations are done on them. Figure 4.2 shows part of the implementation of `dyn_var<T>` type template. Very first, we notice that unlike `static_var<T>`, `dyn_var<T>` does not have a member to store the concrete value or automatic conversion functions. Instead, it has a pointer to a `block` variable AST node. This is stored so that references to this variable can be generated whenever this `dyn_var<T>` is used. The constructor for this type also allocates this new variable and inserts a declaration statement into the generated program $\mathbb{Q}_{SA}$. The variable is given a unique name, and the type AST is generated based on the wrapped type `T`. The `type_extractor<T>::get_type` function generates and returns the AST node corresponding to the type `T` to be later generated in the output code. This handles the first operation on `dyn_var<T>`, which is a declaration. Whenever a variable annotated as *dynamic* is declared in $\mathbb{P}_S$, an equivalent variable in $\mathbb{Q}_{SA}$ is generated. The Figure 4.2 also shows a copy-constructor for `dyn_var<T>` from the primitive type `int`, which creates a declaration with an initialization. These copy constructors also define conversions from `int` when required. Copy constructors for all primitive types are defined in a similar way.

When the `extract_function_ast` is executed, it internally invokes the program $\mathbb{P}_S$ passed as the first argument and constructs the generated $\mathbb{Q}_{SA}$ as it runs. During the execution of the program, a

reference to the builder_context is stored in a global variable current_context so all operators and other builder functions can access this context.

```
namespace builder {
template <typename T>
class dyn_var {
    block::var* v; // Pointer to an AST variable node

public:
    void create_var() {
        v = new block::var();
        v->name = get_unique_name();
        v->type = type_extractor<T>::get_type():
    }

    dyn_var() {
        create_var();
        current_context.create_declaration_statement(v);
    }
    dyn_var(const int &x) {
        create_var();
        current_context.create_declaration_with_init(v, x);
    }
    ...
};
}
```

Figure 4.2: Part of the definition of the dyn_var type template. Unlike static_var<T>, dyn_var<T> doesn't have a concrete value member of type T.

4.2.1 Handling binary and unary operations on *dynamic* variables

The next step in implementing $eval_{static}$ is to handle all kinds of operations on *dynamic* variables and recursively create program nodes in the generated program. We need to detect the operations being performed with dyn_var<T> without any compiler changes. Fortunately, C++ has rich polymorphism support that allows overloading operators to change the behavior of what happens when a particular operation is performed on a custom user-defined type. We can leverage this to detect operations on dyn_var<T> and create appropriate AST nodes from the block namespace. Figure 4.3 shows the definition of some such unary and binary operators. The assignment operator is just a regular binary operator and is allowed to be overloaded in C++. Arbitrary expressions in the generated code are stored in a special type called builder described with the operator. This type contains a single expression AST node - block::expr*. A dyn_var<T> automatically converts to a builder and creates a variable expression referring to the variable being used. Thus, the operations are defined just on the builder type. Since the operations are defined on the builder type, which can hold arbitrary expressions, and these operations return a builder, expression trees are created recursively. The following operators are overloaded in the same way - &&, &, ||, |,^, +, -, *, /, <, >, <=, >=, <<, >>,=, %, !, - (unary), ~ , & (address of), =, [], (), ++, ++ (prefix), –, – (prefix), +=, -=, *=, /=, &=! , |=,^=.

Furthermore, using C++ generics and SFINAE tricks, the overloaded operators are also defined for types that can automatically convert to `builder`, such as constants of primitive types and `dyn_var<T>`.

```
namespace builder {
class builder {
    block::expr *e;

public:
    template <typename T>
    builder(const dyn_var<T>& v) {
        auto ve = new block::var_expr();
        ve->var = v.v;
        e = ve;
    }

    // The assignment operator has to be defined inside the class
    builder operator = (const builder& b1) {
        return current_context.create_binary_assign(b1.e);
    }
};

builder operator + (const builder& b1, const builder& b2) {
    return current_context.create_binary_plus(b1.e, b2.e);
}
...

builder operator ! (const builder& b1) {
    return current_context.create_unary_not(b1.e);
}
}
```

Figure 4.3: Implementation of the `builder` class, which acts as a wrapper for arbitrary expressions and the definition of the overloaded operators on the builder type.

4.2.2 Handling statements containing *dynamic* expressions

Overloading the binary and unary operators on the `dyn_var<T>` and the `builder` types allows BuildIt to recursively capture and generate expressions containing *dynamic* variables. Next, we need to create statements that contain these expressions. The simplest statements are the void statements where the expression is a standalone statement (or is cast to `(void)`). In the `block` namespace's AST representation described in Chapter 3, these are represented as an `expr_stmt`, which is a statement and just has a single expression as a child. Unfortunately, C++ doesn't provide any way to detect when a value is cast to void (standalone expression), and the return value of the expression is simply discarded, and the statements would never be added to the program $\mathbb{Q}_{SA}$. To solve this, we leverage a simple observation that any expression that is not consumed by another expression *must* be a standalone expression statement and should be converted into an `expr_stmt`. As part of the `builder::builder_context` object, we maintain an ordered set of expressions that have been created

but not consumed by other expressions called the **uncommitted list** ($\mathbb{UC}$). Whenever we create a new expression, it is added to $\mathbb{UC}$. Similarly, when an expression is used as an input to another expression, it is removed from $\mathbb{UC}$. Finally, when we reach a point that is definitely the beginning of a new statement, like a variable declaration or the end of the program, we convert all remaining expressions into `expr_stmt`s and add them to the program $\mathbb{Q}_{SA}$.

Figure 4.4 shows a simple program with binary and unary expressions as standalone statements. Figure 4.5 shows the state of the $\mathbb{UC}$ and the generated program $\mathbb{Q}_{SA}$ as the execution goes on.

- When the constructors for `v1`, `v2`, `v3`, `v4`, `v5` are called, the corresponding declaration statements are added to the $\mathbb{Q}_{SA}$ and the $\mathbb{UC}$ is empty.
- When the expressions `v2 * v3` and `v4 / v5` are executed these expressions are added to the $\mathbb{UC}$.
- When the overloaded plus operator is called, the expressions `v2 * v3` and `v4 / v5` are consumed as the parameters and they are removed from the $\mathbb{UC}$. The newly constructed expression `v2 + v3 + v4 / v5` is added to the $\mathbb{UC}$. This is shown in Figure 4.5a.
- When the overloaded assignment operator is evaluated, the RHS of the assignment is consumed as a parameter and hence is removed from the $\mathbb{UC}$, and the assignment expression is added to the $\mathbb{UC}$. This is shown in Figure 4.5b.
- Next when `v2 && v1` is evaluated, it is added to the $\mathbb{UC}$. The previous assignment is still kept in the $\mathbb{UC}$ since we don't know if it would be used as a subexpression later. This is shown in Figure 4.5c.
- When the assignment operator to `v4` is called, the RHS is consumed and removed from the $\mathbb{UC}$ and the new assignment is added. This is shown in Figure 4.5d.
- Finally, when the constructor to `v6` is called, we can guarantee the start of a new statement and all uncommitted expressions in $\mathbb{UC}$ are converted to expr_stmts and added to the $\mathbb{Q}_{SA}$ and removed from the $\mathbb{UC}$. After this, the declaration statement for `v6` is added to the $\mathbb{Q}_{SA}$ and the evaluation terminates. This is shown in Figure 4.5e.

Thus, by keeping track of expressions that haven't been used as sub-expressions to other statements and adding them to the $\mathbb{Q}_{SA}$ only when a new statement starts, we are able to identify standalone statements containing *dynamic* variable even though C++ doesn't allow overloading cast to void.

```
1 void foo() {
2     dyn_var<int> v1, v2, v3, v4, v5;
3     v1 = v2 * v3 + v4 / v5;
4     v4 = v2 && v1;
5     dyn_var<int> v6;
6 }
```

Figure 4.4: Simple program with binary and unary expressions with `dyn_var<T>` as standalone statements.

```
UC = {
    ~~v2 * v3,~~
    ~~v4 / v5~~
    v2 * v3 + v4 / v5
}
```

```
void foo_s() {
   int v1, v2, ...;


}
```

(a) State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ after the RHS of the first statement is evaluated.

```
UC = {
    ~~v2 * v3 + v4 / v5~~,
    v1 = v2 * v3 + v4 / v5


}
```

```
void foo_s() {
   int v1, v2, ...;


}
```

(b) State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ after the first statement is evaluated.

```
UC = {
    v1 = v2 * v3 + v4 / v5,
    v2 && v1
}
```

```
void foo_s() {
   int v1, v2, ...;

}
```

(c) State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ after the RHS of the second statement is evaluated.

```
UC = {
    v1 = v2 * v3 + v4 / v5,
    ~~v2 && v1~~,
    v4 = v2 && v1
}
```

```
void foo_s() {
   int v1, v2, ...;


}
```

(d) State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ after the second statement is evaluated.

```
UC = {
    ~~v1 = v2 * v3 + v4 / v5,~~
    ~~v4 = v2 && v1~~


}
```

```
void foo_s() {
   int v1, v2, ...;
   v1 = v2 * v3 + v4 / v5;
   v4 = v2 && v1;
   int v6;
}
```

(e) State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ after the constructor for **v6** is called.

Figure 4.5: State of $\mathbb{UC}$ and $\mathbb{Q}_{SA}$ at various point of execution of $\mathbb{P}_S$.

4.2.3 Handling conditions containing *dynamic* conditions

The next step in the implementation of $eval_{static}$ is handling if-then-else statements where the condition depends on at least one *dynamic* variable. As explained in Chapter 2, since *dynamic* variables and expressions do not have a concrete value during the execution of $eval_{static}$, we cannot

evaluate whether the then or the else branch would be taken during the execution. **REMS** requires us to create two executions, each taking the separate branches and combining the resultant sub-programs into a single program. However, as explained in Section 3.1.1, due to the limitations of C++, changing the behavior for these if-then-else statements is not possible without changing the compiler since C++ does not allow overloading control-flow constructs. To handle the if-then-else statement, we first need to identify when a *dynamic* variable or expression is used in a condition.
Figure 4.6 shows a simple program with an if-then-else statement where the condition depends on a *dynamic* variable. As the execution of this program begins, the statement for declaration is generated. Next, the expression `x > 13` is created and added to the $\mathbb{UC}$. At this point, according to the C++ semantics, the program tries to convert the expression (of type `builder`) to a `bool` to determine if the expression is *truthy* or *falsy*. For user-defined types like `builder`, it does so by calling the explicit cast to bool operator for the user-defined type. We use this as a way to identify when `dynamic` expressions are used as a condition. Now we need to terminate the execution of the program and create two new executions of $eval_{static}$ E_T and E_F. The simplest way to stop the execution of $\mathbb{P}_S$ is to throw an exception, which can then be caught by `extract_function_ast`. So, as shown in Figure 4.7, we overload the cast to bool operator and throw an exception of type `builder::OutOfBoolsException`. Throwing an exception allows sending the termination message to the calling code through the user program $\mathbb{P}_S$. This exception also carries with it the expression on which the condition was evaluated.

```
1  dyn_var<int> foo (dyn_var<int> x) {
2      dyn_var<int> y = 42;
3
4      if (x > 13) {
5          y = y + 3;
6      } else {
7          y = y - 3;
8      }
9
10     y = y * 2;
11     return y;
12 }
```

Figure 4.6: A simple program with an if-then-else statement where the condition is dependent on a `dyn_var<int>` parameter.

After detecting the start of the if-then-else statement, we need to create two evaluations of $eval_{static}$ that both begin from the same point and take the true and false paths, respectively. This is done through creating two new `builder_context` objects and executing `extract_function_ast`. However, in C++, it is not possible to begin the execution of a program mid-way through the function. We need to begin the evaluations E_T and E_F from the very beginning and reach the same point of execution. In this example, this is the first condition on *dynamic* variables, but in general, for any arbitrary if-then-else, there could be previous if-then-else we had seen, and this execution could already be a recursive execution. As a result, the executions of these two programs must be guided to the same point where we had stopped taking all the same branches we had taken before. To achieve this, we maintain a vector of boolean values in each `builder_context` named `bool_vector` ($\mathbb{BV}$). The $\mathbb{BV}$ holds the exact sequence of boolean values returned to get to the point of the new

```
namespace builder {

struct OutOfBoolsException: public std::exception {
    block::expr *e; // expression on which the condition was used related metadata
};

class builder {
    block::expr *e;

public:
    ...
    explicit operator bool() {
        if (current_context.bool_vector.size() == 0) {
            OutOfBoolsException exp;
            exp.e = e;
            throw exp;
        } else {
            bool ret = current_context.bool_vector[0];
            current_context.bool_vector.pop_front();
            return ret;
        }
    }
};
template <typename T, typename...Args>
auto builder_context::extract_function_ast_impl(T func, std::string fname, Args...args) {
    ...

    try {
        auto res = func(args...);
    } catch (OutOfBoolsException &e) {
        // Condition detected
        // Create two new executions and call extract_function_ast_impl recursively.
        ...
    }
    ...
}
}
```

Figure 4.7: Implementation of the **`OutOfBoolsException`** type, the explicit cast to bool operator, and catching the exception in **`extract_function_ast_impl`**.

if-then-else condition. For the very first execution, this $\mathbb{BV}$ is empty. Whenever a cast to a bool operator is called, we first check if this vector is empty. If it is not empty, we still haven't reached the point of the new if-then-else, and we simply return the next bool from $\mathbb{BV}$. Only when the $\mathbb{BV}$ is empty, we throw the **`OutOfBoolsException`**. When starting the evaluation, two new sub-evaluations, E_T and E_F, the initial $\mathbb{BV}$ of that execution is cloned into the two new **`builder_context`** objects. A bool constant **`true`** is pushed into the context for E_T, and a false constant **`false`** is pushed into the context for E_F, which would cause them to take different paths when the if-then-else condition is reached.

When the two recursive executions finish, the programs $\mathbb{Q}_{STA}$ and $\mathbb{Q}_{SFA}$ that are generated are merged according to the procedure described in **REMS**, and the new program with an if-then-else statement created is returned. Thus, by maintaining a vector of bools in the `builder_context` that describes the path taken on prior conditions, we are able to restart the execution of the program from any arbitrary point in the function without making any changes to the C++ compiler or the runtime.

4.2.4 Implementing Static Tags and Memoization

One of the most important requirements of **REMS** for both correctness and performance is the notion of a static tag. Static tags are a unique identifier attached to each generated expression and statement that tracks the location of the operator in the representation of $\mathbb{P}_S$ from which the expression originated. Furthermore, it also tracks the state of all *static* variables that were live at that point to differentiate two statements that are generated by the evaluation of *static* expressions from the same point. For instance, statements created by a loop with a *static* index. The static tag also uniquely tracks the state of the evaluation of $eval_{static}$ such that if two separate evaluations of $eval_{static}$ on the same program and same inputs generate the same tag at any points, the evaluation and the code generated from that point would be exactly the same. This allows us to implement early merging of if-then-else branches to avoid exponential blowup in extraction complexity and also to detect back edges created by loops.

As explained in Chapter 2, the static tag has two parts. The first part is a log of the point in the program $\mathbb{P}_S$ where the operator was called. If the language supports function calling, this location is the entire sequence of calls that led to the operator. Since C++ has function calls and we would want the developer of $\mathbb{P}_S$ to be able to write complex programs, we need to capture the entire sequence of locations. C++ semantics don't have a notion of precise program location or stack traces, but most C++ implementations support taking a stack trace at any point using a library. This stack trace contains the sequence of instruction pointers (`rip`s) that led to the call. In the implementation of BuildIt, we use libbacktrace [172] to obtain the sequence of `rip`s on the stack. Figure 4.8 shows the implementation of the `static_tag` class and the `gen_static_tag_here` function. The `static_tag` class has a member called `rips` which is a vector of 64-bit values. The `gen_static_tag_here` function calls the `backtrace` function from `libbacktrace` to obtain the entire stack break. It then adds the `rip`s to the newly created tag till the stack trace reaches `start_frame_address`, which is a special frame that is inserted as a guard wrapping $\mathbb{P}_S$. This is to ensure static tags from all evaluations have the same size, even when the `extract_function_ast_impl` function is called recursively.

The second part of the static tag requires us to take a snapshot of all live `static_var<T>` variables when `gen_static_tag_here` is called. We need to keep a list of all such live variables. As shown in Figure 4.9, we augment the constructor of `static_var<T>` to push the address and size of the wrapper value of type `T` into a `live_vars` vector in the `builder_context` object. Similarly, we augment the destructor of `static_var<T>` to remove the address of the wrapped value from the live value list. Finally, in the implementation of `get_static_tag_here`, we iterate through all the entries in `live_vars` from the current context and copy the bytes into the other `static_var_snapshot` member of `static_tag` as shown in Figure 4.8. We also define a comparison operator to compare two static tags by comparing the rips and the bytes recorded. Notice that since we are copying the bytes of the wrapped value and comparing them for byte equivalence, our implementation of `static_var<T>` only works for Plain Old Datatypes (POD types). This is just a limitation of the current implementation, but it can be extended to support wrapping any types as long as they support a copy constructor

and equality operator. This can be achieved by changing the `static_var_snapshot` to be a vector of type-erased containers that can just call the comparison operator on the wrapped value and snapshot it using the copy constructor.

Figure 4.10 shows the updated implementation of the overloaded operators on `builder` and `dyn_var<T>` to call the `gen_static_tag_here` function and attach the tag to the generated expression. This completes the implementation of `static_tag` in BuildIt, which can now be used by $eval_{static}$ for early merging and detecting loops.

To implement memoization for early merging of branches, we simply maintain a memoization map $\mathbb{MM}$ as described in Chapter 2 in the main `builder_context` object as `std::map<static_tag, std::vector<block::stmt*>> memoization_map` This memoization map is passed to all subsequent `builder_context` objects created by a reference so they can share and update it. During the evaluation of $\mathbb{P}_S$ whenever a new statement is generated, the static tag is looked up in this shared map. If it exists, a new type of exception `MemoizationException` is thrown, which is again caught by `extract_function_ast_impl` to stop the evaluation and simply copy over the statements. This map is updated every time as a new program $\mathbb{Q}_{SA}$ from a sub-execution of $eval_{static}$ is returned, as shown in Figure 4.11.

4.2.5 Handling loops on *dynamic* values

The final type of control-flow to handle in the implementation of $eval_{static}$ in BuildIt is loops. We discussed the changes in $eval_{static}$ to handle loops using static tags in Chapter 2. Whenever evaluation of $eval_{static}$ on $\mathbb{P}_S$ reaches a while loop, it is supposed to convert the while loop into an if-then statement with the same condition and body, and add a goto at the end of the loop (and the branches that end in continues) which goes to the if-then. The evaluation of this if-then is handled just like any other if-then-else. Since our handling of if-then-else is done through overloading the cast-to-bool operator on *dynamic* values, we don't need to make any changes. Since in C++, the condition of a while-loop with expressions of user-defined type is also resolved by calling the cast-to-bool operator, our implementation is automatically invoked as if this were an if-then-else. If we end up taking the `true` path, the loop will also automatically go back to the condition at the end of the body, which is the same as having inserted a goto.

The only change we need to make to our implementation of BuildIt is to detect backedges and insert a goto into the generated code as described in Chapter 2. To do this, we maintain a local set in the `builder_context` for the $\mathbb{ML}$ defined as `std::set<static_tag> visited_tags`. Unlike `memoization_map`, this set is not shared with all executions and is unique to each `builder_context` and thus the invocation of $eval_{static}$. Just like with memoization, before inserting a new statement into the body of $\mathbb{Q}_{SA}$, we check if the static tag for this statement exists in the set `visited_tags`. If it does, we once again throw an exception of type `LoopBackException` carrying the tag. When this exception is caught in `extract_function_ast_impl` function, a goto is inserted. The inserted goto simply has the target statement's static tag. Later, during post-processing, labels corresponding to the targets of these gotos are created. However, if the newly generated statement's tag is not present in `visited_tags`, the new tag is inserted into `visited_tag` and the execution continues as usual after inserting the statement into the body of the generated program. Figure 4.12 shows the changes to `add_stmt_to_program` and `extract_function_ast_impl`. Notice that the check for loop back edges is performed before the memoization checks for copying over statements to avoid copying over statements when a loop back edge is necessary.

```
namespace builder {

class static_tag {
public:
    typedef uint64_t rip_t;
    typedef unsigned char byte_t;
private:
    std::vector<rip_t> rips;
    std::vector<byte_t> static_var_snapshot;
public:
    bool operator == (const static_tag& other) {
        if (rips.size() != other.rips.size()) return false;
        if (static_var_snapshot.size() != other.static_var_snapshot.size()) return false;
        for (unsigned i = 0; i < rips.size(); i++)
            if (rips[i] != other.rips[i]) return false;
        for (unsigned i = 0; i < static_var_snapshot.size(); i++)
            if (static_var_snapshot[i] != other.static_var_snapshot[i]) return false;
        return true;
    }
};

struct tracking_tuple {
    char* addr;
    size_t size;
};

static_tag gen_static_tag_here(void) {
    static_tag tag;
    // Obtain the stack trace at this point and add all rips to the tag
    void *buffer[MAX_BUFFER_SIZE];
    int backtrace_size = backtrace(buffer, MAX_BUFFER_SIZE);
    for (int i = 0; i < backtrace_size; i++) {
        if ((rip_t)buffer[i] == start_frame_address)
            break;
        tag.rips.push_back((rip_t)buffer[i]);
    }
    // Iterate through all live static_vars and add their bytes to the snapshot
    for (auto tuple: current_context.live_vars) {
        for (auto i = 0; i < tuple.size; i++) {
           tag.static_var_snapshot.push_back(tuple.addr[i]);
        }
    }
    return tag;
}
}
```

Figure 4.8: Implementation of the **static_tag** class, implementation of the **gen_static_tag_here** function.

Thus, in a program with a while loop, the generated program will have if-then-else conditions with some branches going back to the condition. With these simple changes to $eval_{static}$, we are

```
namespace builder {

template <typename T>
class static_var {
    T val; // The wrapped value stored as a member
public:
    // Register the wrapped value's address and size for snapshotting
    static_var() {
        tacking_tuple t;
        t.addr = (char*)&val;
        t.size = sizeof(T);
        current_context.live_vars.push_back(t);
    }
    // Deregister the wrapped value's address and size when the static_var is deleted
    ~static_var() {
        size_t new_pos = 0;
        for (size_t pos = 0; pos < current_context.live_vars.size(); pos++) {
            if (current_context.live_vars[pos].addr == (char*)&val) continue;
            current_context.live_vars[new_pos++] = current_context.live_vars[pos];
        }
    }
};
}
```

Figure 4.9: Implementation of the constructors and destructors of `static_var<T>` augmented to register and deregister the wrapped value with the current context object.

```
namespace builder {
builder operator + (const builder& b1, const builder& b2) {
    auto ret = current_context.create_binary_plus(b1.e, b2.e);
    ret.e->static_tag = gen_static_tag_here();
    return ret;
}
}
```

Figure 4.10: Changes to the overloaded operators on *dynamic* values to generate the static tag at the point and attach it to the generated expression.

able to extract rudimentary if-then-else and goto loops. In upcoming sections, I will explain the post-processing passes where these are massaged into while loops and potentially for-loops.

4.3 Extracting types for *dynamic* variables

So far, in the description of $eval_{static}$, we discussed how a simple library approach can be used to implement **REMS** in C++ to extract expressions, statements, and control-flow. However, C++ is a typed language. Types not only help with better readability of the code, but also help the compiler

```
namespace builder {
struct MemoizationException: public std::exception {
    static_tag tag; // The tag on which the memoization hit occurred
};

void builder_context::add_stmt_to_program(block::stmt* s) {
    // Check if this statement is in the memoization map
    if (memoization_map.find(s->static_tag) != memoization_map.end()) {
        MemoizationException exp;
        exp.tag = s->static_tag;
        throw exp;
    }
    // if not found push the statement into the generated program
    gen_program.stmts.push_back(s);
}

template <typename T, typename...Args>
auto builder_context::extract_function_ast_impl(T func, std::string fname, Args...args) {
    ...
    try {
        auto res = func(args...);
    } catch (OutOfBoolsException &e) {
        ...
    } catch (MemoizationException &e) {
        // Since we have a memoization hit, complete this program by simply
        // copying over the statements
        for (auto s: memoization_map[e.tag]) {
            gen_program.stmts.push_back(s);
        }
    }
    ...
    // Finally update the memoization_map before returning
    for (auto s: gen_program.stmts) {
        memoization_map[s->static_tag] = gather_statements_after(s, gen_program.stmts);
    }
    return gen_program;
}
}
```

Figure 4.11: Implementation of the **`MemoizationException`** class, the implementation of check while adding a statement to the program to check and throw an exception if it has been memoized, and the changes to **`extract_function_ast_impl`** to catch the exception and copy over the statements, and also update the **`memoization_map`** at the end

optimize the code better to generate a high-performance executable. All the code we generate needs to have types assigned to individual variables and functions. If the input multi-stage program $\mathbb{P}_S$ has user-defined data-types, those data types should be generated in the generated code. Essentially, when the constructor for **`dyn_var<T>`** is called, we are supposed to generate a declaration for the variable in the generated program $\mathbb{Q}_{SA}$. We need a way to extract the AST nodes corresponding

```
namespace builder {
struct LoopBackException: public std::exception {
    static_tag tag; // The tag on which the loop back hit occurred
};

void builder_context::add_stmt_to_program(block::stmt* s) {
    // Check for loopback before memoization
    if (visited_tags.find(s->static_tag) != visited_tags.end()) {
        LoopBackException exp;
        exp.tag = s->static_tag;
        throw exp;
    }
    // Next check for memoization
    if (memoization_map.find(s->static_tag) != memoization_map.end()) {
        ...
    }

    // Update the visited_tags
    visited_tags.insert(s->static_tag);
    gen_program.stmts.push_back(s);
}

template <typename T, typename...Args>
auto builder_context::extract_function_ast_impl(T func, std::string fname, Args...args) {
    try {
        auto res = func(args...);
    } catch (OutOfBoolsException &e) {
        ...
    } catch (MemoizationException &e) {
        ...
    } catch (LoopBackException &e) {
        auto gs = new block::goto_stmt();
        gs->target_static_tag = e.tag;
        gen_program.stmts.push_back(gs);
    }
    ...
}
}
```

Figure 4.12: Changes to the implementation of **`add_stmt_to_program`** and **`extract_function_ast_impl`** to handle loopback edges

to the type **`T`** so that they can be printed out. As shown in Figure 4.2, the behavior of **`dyn_var<T>`** doesn't really depend on the type **`T`** it wraps around. This is because all the operators are overloaded in the same way, regardless of the type. The only place where this type is used is in the constructor of **`dyn_var<T>`**, where we call **`type_extractor<T>::get_type()`** to get the corresponding AST nodes for the type.

There are three main kinds of types we need to extract. The first is primitive types present in C++ like **`int`**, **`long`**, **`char`**, and so on. These types are part of the C++ language itself and need

to be generated as it is. For the primitive types, we simply implement a partial specialization for the **type_extractor** class to return the appropriate type as shown in Figure 4.13. Since the list of primitive types in C++ is finite and small, we define such manual specializations for **short**, **int**, **long**, **long long**, **char**, **float**, **double**, **bool**, **void**, **long double**, and their **unsigned** versions wherever applicable. Other less frequently used primitive types like **wchar_t** or **char16_t** can be handled by the user using **named_types** described later. We handle the most common ones for convenience.

```
namespace builder {
// Primary template for the type_extractor helper class

template <typename T>
struct type_extractor;

// Partial specialization to extract the int type
template <>
struct type_extractor<int> {
    // the get_type function creates a scalar type node
    // with the scalar type id set to int
    block::type* get_type() {
        auto st = new block::scalar_type();
        st->scalar_type_id = scalar_type::INT_TYPE;
        return st;
    }
};

// Partial specialization to extract the unsigned short type
template <>
struct type_extractor<unsigned short> {
    // the get_type function creates a scalar type node
    // with the scalar type id set to short int with the unsigned qualifier added
    block::type* get_type() {
        auto st = new block::scalar_type();
        st->scalar_type_id = scalar_type::SHORT_TYPE;
        st->qualifiers |= scalar_type::UNSIGNED;
        return st;
    }
};

// Manual specialization for the rest of the primitive types follows

}
```

Figure 4.13: Partial specialization implementation of the **type_extractor** class for the primitive types in C++.

The next kind of types includes compound types, which contain arrays, pointers, references, and function types of other types. We once again define partial specialization for the **type_extractor** helper class to handle these. These specializations call the **type_extractor** recursively on the inner types to handle arbitrary levels of compounding of types. Figure 4.14 shows the partial specializations for pointer, array, and array types. The specialization for reference types is defined

similarly to the pointer types.

```cpp
namespace builder {
// Primary template for the type_extractor helper class

template <typename T>
struct type_extractor;

// Partial specialization to extract pointer types
// the template parameter T is the inner type here
template <template T>
struct type_extractor<T*> {
    // the get_type function creates a pointer type node
    // and recursively gets the pointee type
    block::type* get_type() {
        auto pt = new block::pointer_type();
        pt->pointee_type = type_extractor<T>::get_type();
        return pt;
    }
};

// Partial specialization to extract array types
template <typename T, size_t size>
struct type_extractor<T[size]> {
    // the get_type function creates an array type node
    // and recursively gets the element type
    block::type* get_type() {
        auto at = new block::array_type();
        at->element_type = type_extractor<T>::get_type();
        at->size = size;
        return at;
    }
}


// Partial specialization to extract function types
template <typename RetType, typename...ArgsTypes>
struct type_extractor<RetType(ArgTypes...)> {
    // the get_type function recursively gets the return type
    // and argument types
    block::type* get_type() {
        auto ft = new block::function_type();
        ft->return_type = type_extractor<RetType>::get_type();
        ft->arg_types = std::vector<block::type*>({type_extractor<ArgTypes>::get_type()...});
    }
}
```

Figure 4.14: Partial specialization implementation of the **`type_extractor`** class for pointer, array and function types in C++.

The last kind of type to handle is a custom type of user-defined types that are defined as part of

the developer's implementation in $\mathbb{P}_S$. The most common user-defined type is the `struct` type. We implement two ways to handle user-defined types.
The simplest user-defined types just have a name that needs to be generated wherever they are used. This is common for opaque types from libraries to be linked with the generated $\mathbb{Q}_{SA}$, and the members of the type aren't relevant. For example, the `FILE` type from the standard C library is always used as a handle for an opened file as `FILE*` and the members or implementation of `FILE` don't matter for the execution of $\mathbb{P}_S$. To use such types, BuildIt provides the `builder::name` template. The `name` template accepts a `const char*` (address of a global variable of type array of char) as a template parameter and uses that as the name of the type. The `type_extractor` class handles this with a specialization. Figure 4.15 shows the definition of the `name` template, the partial specialization, and how named types are used in $\mathbb{P}_S$.

Notice that the `builder::name` template also accepts template arguments for the named type as extra arguments after the string name. This can be used to generate named opaque template types like `std::vector<int>`. BuildIt's implementation of named types deviates slightly from the pure type-based model, where some seamlessness is sacrificed. This is, however, required since the developer has to provide information about the type of name to be generated.

4.3.1 Handling non-opaque user-defined types for *dynamic* variables

In the cases where the developer doesn't care about the implementation of the user-defined type in $\mathbb{P}_S$ or the $\mathbb{Q}_{SA}$, the developer can use `builder::name`. However, in many cases, the members (functions and variables) of the user-defined types are used as part of the implementation and should be faithfully extracted. We need to handle not only the name and the declaration of the variables but also the expressions where the members are accessed as part of the implementation of $\mathbb{P}_S$ with as seamless an interface as possible.

Figure 4.16a shows a simple user-defined type `struct cylinder` with two members `radius` and `height`, each of which is a floating point value, and Figure 4.16b shows the same program with `dyn_var<T>` added to all parameters, return types, and variables. Besides not having access to any information about the type, there is a problem with the expressions `c.rad` and `c.height`. Since our `dyn_var<T>` variables are purely placeholders, they don't actually have the right members, and hence these expressions aren't syntactically correct. To make these correct, we would have to add the members to the definition of `dyn_var<struct cylinder>`. This could be done automatically with the proposed *Type Reflection* feature in C++26, but as of today, the C++ standard doesn't support this. We would somehow have to inject the members `rad` and `height` with C++11 available features.
The simplest way to insert members of `struct cylinder` into `dyn_var<struct cylinder>` is to make the latter inherit from the former. Figure 4.17 shows the definition of `dyn_var<T>` when `T` is an aggregate type that is not one of BuildIt's own types. This check is required to avoid erroneous inheritance when `T` is not an aggregate type like `int` or a pointer.

With this change to the definition of `dyn_var<T>`, we solve two problems. First, variables of type `dyn_var<struct cylinder>` already have the members `rad` and `height` present, making the expressions `c.rad` and `c.height` syntactically valid. Second, these variables are also of `dyn_var<float>` type, which means their operators are already overloaded and they would behave like any other *dynamic* values. However, we run into another problem, which is that since these members are just regular `dyn_var<T>`, they will generate standalone variable declarations alongside the declaration for `c` since the constructors are called in order. We need to tell these members that they are not standalone

```
namespace builder {

template <const char* N, typename...Args>
class name {}; // The name class itself is empty

template <const char* N, typename...Args>
class type_extractor<name<N, Args...>> {
    // Template arguments, if any, to the named type are
    // supplied as arguments after the name
    block::type* get_type() {
        auto nt = new block::named_type();
        nt->type_name = N;
        // Extract template arguments if there are any
        nt->template_args =
            std::vector<block::type*>({type_extractor<Args>::get_type()...});
        return nt;
    }
};
}
...
// Program using named type

// Definition of the opaque type FILE
static const char file_t_name[] = "FILE";
using MY_FILE = builder::name<file_t_name>;

// Definition of the opaque named type template std::vector
static const char my_vector_t_name[] = "std::vector";
template <typename T>
using my_vector = builder::name<my_vector_t_name, T>;

void foo (void) {
    // Use a pointer to the opaque type FILE
    dyn_var<MY_FILE*> f = my_fopen("myfile.txt", "r");

    // Use the opaque template type std::vector<T>
    dyn_var<my_vector<int>> x = {1, 2, 3};
}
```

Figure 4.15: Implementation of `builder::name` template, the specialization for `type_extractor` and a program using a named type

variables but members of another `dyn_var<T>`. We could use a separate type like `dyn_member<T>` or a copy constructor helper for `dyn_var<T>` that would 1. suppress the generation of a separate declaration and 2. refer to the parent variable when used and generate a member access expression (`block::member_access_expr*`. However, both these approaches would require changes by the user, compromising seamlessness.

To solve this, let us look at the order in which the constructors for the three *dynamic* variables are called. First, the constructor for `c` (`dyn_var<struct cylinder>`) is called. Next, the constructor

```
struct cylinder {
   float rad;
   float height;
};
float volume(
       struct cylinder c){
   float v =
       3.14 * (c.rad * c.rad) * c.height;
   return v;
}
```

(a) A simple user-defined type in C++ and a function using the type.

```
struct cylinder {
   dyn_var<float> rad;
   dyn_var<float> height;
};
dyn_var<float> volume(
       dyn_var<struct cylinder> c){
   dyn_var<float> v =
       3.14 * (c.rad * c.rad) * c.height;
   return v;
}
```

(b) Program with user defined type with `dyn_var<T>` types added to all parameters and variables.

```
namespace builder {

// Check if T is a class, isn't a dyn_var<T> or a static_var<T> itself
template <typename T>
constexpr bool is_non_buildit_aggregate = ...;

template <typename T, typename Enable=void>
struct dyn_var_parent_provider {}; // Empty parent in case T is not an aggregate

template <typename T>
struct dyn_var_parent_provider<T, std::enable_if_t<is_non_buildit_aggregate<T>>>
    : public std::remove_reference<T>::type {}; // If T is an aggregate inherit from T

// Insert the right parent in dyn_var
template <typename T>
class dyn_var: public dyn_var_parent_provider<T> {
    ...
};

}
```

Figure 4.17: Changes to the implementation of `dyn_var<T>` to inherit from `T` when `T` is a non buildit aggregate type.

`struct cylinder` is called, which recursively calls the constructor of `rad` (`dyn_var<float>`) and `height` (`dyn_var<float>`) in that order. The actual body of the constructor for `dyn_var<struct cylinder>` is called after the members are initialized. The C++ standard [50] specifies the order of evaluation of constructors as follows -

> The order of member initializers in the list is irrelevant, the actual order of initialization is as follows:
>
> 1. If the constructor is for the most-derived class, virtual bases are initialized in the order in which they appear in depth-first left-to-right traversal of the base class declarations (left-to-right refers to the appearance in base-specifier lists).
> 2. **Then, direct bases are initialized in left-to-right order as they appear in this class's base-specifier list.**

3. Then, non-static data member are initialized in order of declaration in the class definition.
4. Finally, the body of the constructor is executed.

** emphasis added*

According to the second point, if a class inherits from multiple classes, the constructors of the base classes are called left-to-right in the order they appear in the base-specifier list. We can leverage this to convey to the inherited members information about the parents. We wrap the appearance of `T` in the base specifier-list for **`dyn_var_parent_provider`** between two special classes **`member_begin<T>`** and **`member_end`**. The **`member_begin<T>`** records the address of the parent variable onto a global stack. When any **`dyn_var<T>`** is constructed, it first checks this stack. If the stack is empty, it creates a declaration for the variable and gives the variable a unique name. But if the stack is not empty, it takes the address at the top of the stack and remembers it as the parent. When these *dynamic* variables are used in any expression, they produce a member access expression instead of a var expression. Figure 4.18 shows the implementation of **`member_begin<T>`** and **`member_end`** and the changes to the base-specifier list of **`dyn_var_parent_provider`**. Figure 4.19 shows the changes to **`dyn_var<T>`** where each variable has a mode where it can be **`STANDALONE`** or a **`MEMBER`**. The constructor has been updated to check the parent stack to initialize the **`dyn_var<T>`** in the appropriate mode. When this variable is used in any expression (and implicitly converted to **`builder`**), the mode is checked to produce expressions of either **`block::var_expr`** or either **`block::member_access_expr`** type.

```
namespace builder {
// a common base class for all dyn_vars
class dyn_var_base {
    virtual block::expr* get_access_expr();
};

std::vector<dyn_var_base*> parent_stack; // Global parent stack starts out as empty
template <typename T>
struct member_begin {
    member_begin() {
        // Push _this_ on the stack after performing appropriate casts
        parent_stack.push_back(static_cast<dyn_var_base*>(static_cast<dyn_var<T>*>(this)));
    }
};
struct member_end {
    // Pop off _this_ from the stack once all members are done
    member_end() { parent_stack.pop_back(); }
};

// Wrap the inheritance from T between member_begin and member_end
template <typename T>
struct dyn_var_parent_provider <T, std::enable_if_t<is_non_buildit_aggregate<T>>>
    : public member_begin<T>, public std::remove_reference<T>::type, public member_end {};
```

Figure 4.18: Implementation of **`member_begin<T>`** and **`member_end`** and changes to **`dyn_var_parent_provider`**.

```
// Add a check and mode inside the dyn_var
template <typename T>
class dyn_var: public dyn_var_base, public dyn_var_parent_provider<T> {
    enum class mode_t {STANDALONE, MEMBER} mode;
    block::var *v; // Used if mode is standalone
    dyn_var_base* parent; // Used if mode is member
    std::string member_name; // Used if mode is member
public:
    dyn_var() {
        // create variable and declaration if the parent stack is empty
        if (parent_stack.empty()) {
            create_var();
            mode = mode_t::STANDALONE;
            current_context.create_declaration_statement(v);
        } else {
            //Don't create a declaration, but remember the parent
            mode = mode_t::MEMBER;
            parent = parent_stack.back();
            member_name = generate_next_member_name(parent);
        }
    }
    // Function to convert a variable into an expression when used
    block::expr* get_access_expr() override {
        // If mode is standalone, create a var expression
        if (v.get_mode() == mode_t::STANDALONE) {
            auto ve = new block::var_expr();
            ve->var = v.v;
            return ve;
        } else {
            // If mode is member, produce a member access expression,
            // by getting expression from parent
            auto me = new block::member_access_expr();
            me->member_name = member_name;
            me->parent_expr = parent->get_access_expr();
            return me;
        }
    }
};
class builder {
public:
    template <typename T>
    builder(const dyn_var<T>& v) {
        e = v.get_access_expr();
    }
...
};
}
```

Figure 4.19: Changes to the implementation of `dyn_var<T>` to have a mode where the variable can either be standalone or a member of another variable. The constructor and use of these variables have been changed.

```
using builder::with_name;
struct cylinder {
    // Assign a name to the struct by adding a member type_name
    static constexpr const char* type_name = "struct cylinder";
    // Assign names to the members with with_name
    dyn_var<float> rad = with_name("rad");
    dyn_var<float> height = with_name("height");
};
dyn_var<float> volume(dyn_var<struct cylinder> c){
    dyn_var<float> v =
        3.14 * (c.rad * c.rad) * c.height;
    return v;
}
```

Figure 4.20: Updated cylinder type definition and program to assign names to the type and its members.

With these changes, BuildIt is able to extract type information and member access expressions from *dynamic* variables of user-defined type. The above changes generate a type declaration with a unique generated name and unique member names. Furthermore, if the developer needs to generate specific names for the type (for compatibility with external libraries), they can create a special static member inside the user-defined type `static constexpr const char* type_name = "..."`. BuildIt detects the presence of this member using SFINAE tricks and uses the name if present instead of using an auto-generated name. Finally, if the developer needs a specific name for any of the members, they can use the `builder::with_name` copy constructor helper as a default initializer for the member. BuildIt then uses the supplied name for the member. `builder::with_name` is also a generic copy constructor helper and can be used to assign specific names to variables, as explained in the next section. Figure 4.20 shows a version of the modified program with names added. Although this version is less seamless than the original version, it is used only in specific scenarios when external compatibility is required.

4.4 Post-Processing and Code Generation Passes

Once the execution of all $eval_{static}$ have concluded and BuildIt has extracted all expressions, statements, user-defined types as program nodes represented as ASTs in the `block` namespace, BuildIt runs a sequence of passes to ensure correctness of the code and massaging it into more readable code before finally lowering it into C++ source code for $\mathbb{Q}_{SA}$ to be compiled and run in $eval_{dynamic}$. We call most of these passes semantic-preserving passes because they don't change the behavior of the generated program. These passes are all implemented under the `block` namespace, and while most run automatically, some passes need to be enabled or manually invoked. We will discuss the passes here in detail in the order they may run:

4.4.1 Var Namer

Since BuildIt is implemented in a lightweight way as a pure library in C++, it doesn't have access to the declared names of local, global, and parameter variables of type `dyn_var<T>`. As a result, BuildIt generates unique names for variables as they are created. However, since the $eval_{static}$ procedure runs the same program $\mathbb{P}_S$ multiple times and combines their results, the variables created in different runs would have different names, and combining the programs wouldn't produce syntactically correct code. For example, the same variable may appear with different names under the then and else branches of a generated if-then-else condition because the bodies of those statements are generated in different runs. To reconcile variables that have different names, but are logically the same, we uniquely identify variables by the static tag generated when the variable was declared. The var namer then assigns one unique name to variables that have the same static tag. Since this pass is required for generating syntactically correct code, it runs unconditionally. Variables that have user-assigned names using the `with_name` constructor helper are not touched by the var namer pass since the exact user-supplied name might be required. Arguments to functions are assigned names of the form `arg0, arg1, ... argN` in the order they appear in the function. Local variables are given the names `var0, var1, ... varN` in the order they appear in the body of the function. User-defined types are given names `custom_struct0, custom_struct1, ... custom_structN`, in the order they are used, and if they don't have user-supplied names. Finally, members inside a struct are given names of the form `mem0, mem1, ... memN` in the order they appear inside a user-defined struct for the members that don't have user-supplied names.
Furthermore, if the program $\mathbb{P}_S$ itself is compiled with debug symbols (with the `-g` gcc/clang flag) and BuildIt itself is compiled with the `RECOVER_VAR_NAMES=1` config option, the var namer pass accesses the debug information stored inside its own executable and assign names that reflect the name of the original variable. The variables are still assigned a unique integer prefix to disambiguate variables created from the same source location but from different iterations of static variables.

4.4.2 Label Collector and Label Inserter

Recall in Section 4.2.5 we described how a hit in the lookup in the `visited_tags` set leads to a `LoopBackException` which results in the insertion of a `goto` statement with the target statement's static tag. However, the program isn't syntactically correct till the jump target label is actually created and tied to the goto statement(s). The Label Collector and Label Inserter pass first iterate through the generated program and gather all the static tags that are potential jump targets. The passes then iterate through the program again and find the statements with all the tags and insert labels before them. Finally, these label names are stitched into the goto statements based on the static tags they jump to. Once again, since these passes are required for syntactic correctness, they are enabled by default.

4.4.3 Sub Expression Cleaner

Sub-expression cleaner is a simple cleanup pass for removing redundant expressions generated as standalone statements that are also part of other expressions. Remember that expressions are promoted to standalone statements only if they are not consumed by other statements while they are in the $\mathbb{UC}$. But due to specific quirks of C++ return value optimizations, a new statement may begin

even when an old sub-expression hasn't been completely consumed. In this case, the sub-expression is already added to the program $\mathbb{Q}_{SA}$ from the $\mathbb{UC}$. When they are eventually consumed as part of a bigger expression, BuildIt identifies it but just marks the created standalone statement for clean up later. The sub-expression cleaner simply goes through all statements and removes statements that have been marked for cleanup. Since this pass is also required for correctness, it runs unconditionally. Recall that if a statement that a side-effect is inserted redundantly twice, it might lead to correctness issues.

4.4.4 Redundant Copy Elimination

When `dyn_var<T>` variables are passed to functions as part of the implementation of $\mathbb{P}_S$, the copy constructor of the parameter is invoked and a copy of the variable is created. Similarly, a copy is also created when a `dyn_var<T>` variable is returned from a function. With programs using a lot of abstractions, when these function calls are inlined as part of $\mathbb{Q}_{SA}$, a lot of redundant copies of variables might be created. The redundant copy elimination pass, or `RCE`, removes local variables that are never modified and are simply copies of other variables. All the uses of these variables are replaced with the variables they are assigned from. If any side effect appears between the creation of the variable and its uses, the variable is not eliminated. This pass eliminates a lot of redundant copies, simplifying the generated code for readability. This pass only runs when `builder_context` has the `run_rce` flag set to `true`. This flag defaults to `false`.

4.4.5 While Loop Finder

The loop extraction procedure we described in Section 4.2.5 always extracts if-then-else and goto loops. Although these loops are syntactically and semantically correct, they are not very readable in the generated code. The while loop finder converts blocks of code with these gotos into while loops by identifying all back jumps to a target and inserting breaks or continues to preserve the original control flow. Heuristics and specific pattern matching are used to identify the condition and body of the while loop. If the condition cannot be identified, a `while (1)` loop with breaks and continues is generated. This pass runs by default but can be disabled by setting the `feature_unstructured` flag in the `builder_context` to `true` to retain gotos instead of loops.

4.4.6 For Loop Finder

The For Loop finder passes further massages while loops into canonical for loops wherever possible to further improve the readability of the generated code. While loops that have conditions on a single variable that is initialized just before the while loop and is updated on every back-edge in the same way are converted into a for loop. Similar to the while loop finder, this pass runs by default but can be disabled by setting the `feature_unstructured` flag in the `builder_context` to `true`.

4.4.7 If Switcher

Since the if-then-else conditions in the generated code are extracted from scratch by re-execution, the generated statements might not be in the canonical form. Some statements are generated with an empty then branch and an else branch with non-zero statements. The if switcher pass converts

```
auto ast = context.extract_function_ast(...);
// construct a pattern for e + 1
auto pattern = expr("e") + int_const(0);
auto replace_pattern = expr("e");
auto matches = find_all_matches(pattern, ast);
for (auto x: matches) {
    replace_match(ast, x, replace_pattern);
}
```

Figure 4.21: Pattern matching example for replacing $e + 0 \longrightarrow e$ post extraction of the program.

these to canonical conditions by switching the then and else branches and negating the conditions to make the conditions more readable. This pass also runs by default but can be disabled by setting the `feature_unstructured` flag in the `builder_context` to `true`.

4.4.8 Loop Re-Roller

The Loop Re-Roller pass allows the user to roll similar statements into a loop instead of multiple statements. Typically, when a loop with `static_var<T>` index is executed, the loop is completely executed and unrolled as part of the execution to generate multiple statements. These statements might be specialized based on the body of the loop, but could have similar parts. To make the generated code more readable, the user can annotate these statements to be rolled up into a loop. The loop re-roller pass finds similar statements annotated to be rolled up into a loop and identifies the common portions. The portions of the statements that are not common and are specifically different constants are converted into array accesses, and the lookup arrays are generated before the loop. This pass allows converting long repeated statements into statements with array lookups. While this pass runs unconditionally, it is only applied to statements annotated by the user.

4.4.9 User Defined Passes

After all the passes defined as part of the `block` namespace are run, the transformed AST is returned to the user. The user can then run their own analysis and transformation passes to implement any domain-specific transformations. BuildIt provides the infrastructure support for writing visitors and rewriters over the AST nodes for easy analysis and transformations. Furthermore, BuildIt also provides a pattern matcher and replacer interface to quickly replace arbitrary program DAGS of expressions with other expressions. The rewriter library allows for name binding to enforce equality of sub-expressions and also to reuse expressions in the newly inserted expressions. For example, simple rewrites like $e + 0 \longrightarrow e$ where e is an arbitrary expression can be performed with a single line of code. This rewriter API allows non-compiler expert BuildIt users to perform simple transformations without having to understand how ASTs, visitors, and rewriters work. Figure 4.21 shows the API and an example to look for all expressions of the form $e+0$ and replace all occurrences with the corresponding e.

4.4.10 C and C++ Code Generator

The final pass in the execution of BuildIt is the C or C++ Code Generator. This pass is invoked manually by the user and accepts an AST and an output stream. The pass then generates the C or C++ source code for the extracted program into the output stream based on the configuration options. This allows the user to actually generate, compile, and run the output program $\mathbb{Q}_{SA}$. This pass also has API functions to output the user-defined types (structs) to the generated code since these types might be generated by BuildIt with unique type and member names. This pass can further be extended by implementations if they want to generate code for different, closely related languages. For example, an extension BuildIt already exists to generate CUDA as detailed in Chapter 5.

4.5 Supporting External Libraries

The developer translates a given program $\mathbb{P}$ in C++ into an equivalent $\mathbb{P}_S$ given an S, by modifying the source code to add the `dyn_var<T>` and `static_var<T>` types. However, not all the code might be available or might not require the specialization and performance improvement offered by BuildIt. One of the most powerful features that makes writing high-performance code easy is the use of optimized external libraries. I will discuss in this section allowing support for external libraries in both the *static* and *dynamic* stages. For the sake of this discussion, we define external libraries as some code that we either don't have access to the source code of, or don't wish to specialize using BuildIt and use the compiled library as it is. This can include the data types and functions from the external library.

4.5.1 External Libraries in the *static* stage

We define external libraries in the *static* stage as invoking functions from external libraries with values that depend entirely on *static* values, and thus the function can be entirely evaluated in the first stage. Remember that the `static_var<T>` variables have concrete values during the evaluation of $eval_{static}$. So functions from external libraries can be called directly by simply calling the functions as usual C++ code and passing the concrete values from the *static*. As long as the execution of the external function itself doesn't require evaluating any *dynamic* code (by means of lambdas, or function pointers), the execution can be entirely evaluated and the result can be wrapped back into `static_var<T>` variables.
However, there is a challenge – recall that a BuildIt program can have two types of variables with concrete values during the execution of $eval_{static}$. Either variables declared with type `static_var<T>` or regular C++ values that do not mutate during execution. But calling external functions violates this constraint, since the function might have local variables that are mutated during the execution (not declared `static_var<T>`), which would risk incorrect or undefined behavior. However, recall that the mutations to `static_var<T>` are observed by BuildIt only when an overloaded operator on a `dyn_var<T>` variable is called and a static tag containing snapshots of live `static_var<T>` variables is generated. If a variable that is not marked `static_var<T>` is alive only between the call to the overloaded operations `dyn_var<T>` (*dynamic* code), it is not even observed by BuildIt and does not violate correctness.

We thus relax the constraint that a *static* variable that is declared with regular C++ can be

mutated during the execution of $\mathbb{P}_S$ if and only if no overloaded operators on `dyn_var<T>` variables are called during its lifetime. Care must be taken to ensure that if the external library has any global state or handles to objects that live across a function call, they must be constant or declared `static_var<T>`.

This constraint also exists only on functions called from the function passed to `extract_function_ast` ($\mathbb{P}_S$). Arbitrary functions can be called before and after the call to `extract_function_ast`, and results can be passed into $\mathbb{P}_S$. Figure 4.22 shows a call to the popular Eigen [81] library to implement a function that accepts an angle of rotation and a displacement vector and applies the combined transformation to an input point. In this example, we specialize the implementation to the given angle and displacement vector, i.e., the angle and vector are *static* parameters and the actual point to transform is *dynamic*. We can notice that this function has several external library calls, all depending on the first stage. First, we call the `cmath` functions `sin` and `cos`. We then call the construct the rotation matrix by calling functions from the Eigen library. We do the same for the translation transformation. Finally, we combine the two transformations by invoking the matrix multiplication operation from Eigen. The final result of the computation is then used to apply the transformation to the input point. When this program is executed with BuildIt with arguments for angle and translation vector, the output program $\mathbb{Q}_{SA}$ just contains six multiply and add calls to produce the output. Notice that the internal implementation of matrix product, `sin`, `cos`, all have local variables being mutated. But all those mutations happen between any *dynamic* operations.

4.5.2 External Libraries in the *dynamic* stage

We define external libraries in the *dynamic* stage as invoking functions from external libraries with values that depend on at least one *dynamic* value. The implementation of this function also needs to be available as a linked library in the generated program $\mathbb{Q}_{SA}$. Since the function itself is available during $eval_{dynamic}$, all we have to do is generate a call to the external function as part of the generated program. We also want to be able to do this in a seamless way from the first stage. Recall that for user-defined types in C++, the function call operator can be overloaded just like other binary and unary operators. To insert a function call in the generated code, we require the user to declare the function object itself as a `dyn_var<T>` variable where `T` is a function type. Calling the overloaded function call operator on this `dyn_var<T>` simply creates program nodes of type function call, just like the other operators. However, the name of the function target cannot be auto-generated and needs to match the exact name of the function. So we require the developers to assign a name to the `dyn_var<T>` variable using the `builder::with_name` copy constructor helper. Figure 4.23 shows an implementation of insertion and deletion of values in a linked list. Notice that the memory for the linked list node needs to be allocated and freed in the *dynamic* stage by inserting a call to `malloc` and `free`, respectively. We define variables of type `dyn_var<void*(size_t)>` and dynvoid(void*) and assign them names `malloc` and `free`. We can insert calls to these functions by simply "calling" (invoking the overloaded call operator) these variables with the appropriate parameters. There are no restrictions on the values passed to these functions. If *dynamic* variables are passed, the right variable name is inserted in the call. If a *static* value is passed, a constant with the appropriate value corresponding to the current concrete value of the *static* variable is inserted. Thus, by simply wrapping `dyn_var<T>` around function types, we can support arbitrary optimized external libraries without having to migrate their implementation to C++.

```
#include <Eigen/Dense>
#include <cmath>
void transform_point (const double rotate_angle, const double translate_vector[2],
        dyn_var<double[2]> point) {
    // Create a homogenous point from the input point
    dyn_var<double[3]> point_homogenous = {point[0], point[1], 1.0};

    // Create a transformation matrix for applying rotation
    Eigen::Matrix3d rotation;
    rotation << cos(rotate_angle), -sin(rotate_angle), 0,
                sin(rotate_angle),  cos(rotate_angle), 0,
                0,           0,          1;
    // Create a transformation matrix for application translation
    Eigen::Matrix3d translate;
    translate << 1, 0, translate_vector[0],
                 0, 1, translate_vector[1],
                 0, 0, 1;

    // Obtain a combined transformation, rotate followed by translate
    // by invoking an external library function
    Eigen::Matrix3d transform = translate * rotation;

    // Apply transformation using matrix * vector multiplication unrolled
    for (static_var<int> i = 0; i < 2; ++i) {
        point[i] = 0.0;
        for (static_var<int> j = 0; j < 3; ++j) {
            point[i] += transform(i, j) * point_homogenous[j];
        }
    }
}
```

Figure 4.22: Invoking external functions in *static* stage in BuildIt

4.6 Generalizing to Multiple Stages

To handle generalization to multiple stages, BuildIt simply has to allow wrapping `dyn_var<>` around `dyn_var<T>` and `static_var<T>`. The only change that needs to be made is that these types need to be extracted. The behavior of these variables doesn't actually differ from any `dyn_var<T>`. This is simply implemented by adding a specialization to `type_extractor` for BuildIt types as shown in Figure 4.24. Finally, the C and C++ Code generator handles these types to print out the templated types `dyn_var<T>` and `static_var<T>`.

```
struct node {
    dyn_var<int> value;
    dyn_var<struct node*> next;
};

namespace runtime {
// Define a dynamic version of malloc and free by wrapping dyn_var
// around function types and assigning names
dyn_var<void*(size_t)> malloc = with_name("malloc");
dyn_var<void(void*)> free = with_name("free");
dyn_var<size_t(void)> size_of = with_name("sizeof");
}

/* Both insert_node and delete_nod functions assume
The linked list has a sentinel element in the beginning */

void insert_node (dyn_var<struct node*> list, dyn_var<int> value) {
    dyn_var<struct node*> new_node;
    new_node = runtime::malloc(runtime::size_of(*new_node));

    new_node->value = value;
    new_node->next = nullptr;
    // Seek to the end of the list
    while(list->next != nullptr) list = list->next;
    // Insert the new node
    list->next = new_node;
}

void delete_node(dyn_var<struct node*> list, dyn_var<int> value) {
    while (list->next != nullptr) {
        if (list->next->value == value) {
            dyn_var<struct node*> todelete = list->next;
            list->next = list->next->next;
            runtime::free(todelete);
        }
    }
}
```

Figure 4.23: Implementation showing insertion and deletion of nodes in a *dynamic* linked list caling external functions **`malloc`** and **`free`**

```cpp
namespace builder {

// Partial specialization to extract dyn_var types
// the template parameter T is the inner type here
template <template T>
struct type_extractor<dyn_var<T>> {
    // the get_type function creates a buildit type node
    // and recursively gets the wrapped type
    block::type* get_type() {
        auto pt = new block::buildit_type();
        pt->type_id = DYN_VAR;
        pt->wrapped_type = type_extractor<T>::get_type();
        return pt;
    }
};


// Partial specialization to extract static_var types
// the template parameter T is the inner type here
template <template T>
struct type_extractor<static_var<T>> {
    // the get_type function creates a buildit type node
    // and recursively gets the wrapped type
    block::type* get_type() {
        auto pt = new block::buildit_type();
        pt->type_id = STATIC_VAR;
        pt->wrapped_type = type_extractor<T>::get_type();
        return pt;
    }
};
```

Figure 4.24: Partial specialization implementation of the **`type_extractor`** class for BuildIt types themselves to support more than 2 stages

Chapter 5

BuildIt Extensions for High-Performance Libraries and DSLs

So far in Chapter 2, I described **REMS**, a novel methodology for implementing multi-staging in imperative languages in the presence of cross-stage side-effects. In Chapter 3 and Chapter 4, I described BuildIt, a framework that brings the capabilities of **REMS** to C++ and its implementation in a lightweight type-based way. BuildIt provides the basic primitives to separate the execution of the program into two or more stages using the `dyn_var<T>` and `static_var<T>` type templates. However, simply dividing the execution of a program into stages is often not enough to obtain high performance. The execution of the first stage needs to perform a variety of optimizations and, in the process, ends up simplifying, specializing, fusing, analyzing, and parallelizing the generated code while also moving around data structures where necessary. Typically, in a single-stage program, such operations and optimizations are expensive because they add runtime overhead. However, with multi-staging, a lot of the computation that does the analysis and transformations can be evaluated in the first stage to generate very efficient code.
In this chapter, I will describe practical extensions to the BuildIt system, either built entirely on top of the type-based interface of BuildIt or as a direct extension to the library itself that serve as basic blocks for optimizations in DSLs and high-performance alike.

More concretely, I will describe seven different optimization methodologies that **simplify**, **specialize**, `fuse`, `analyze`, `parallelize`, `organize (data)`, and `hoist` operations to improve the overall performance of the generated code.

5.1 High-Performance Library DSL Example

Before I describe the extension and abstractions for high-performance, I will introduce a C++ embedded DSL called BArray (BuildIt Array DSL) that lets users operate on entire arrays at a time. For the entirety of this chapter, I will assume a scenario where a domain-expert developer is writing this DSL as a library to be used by end-users from the domain. The developer first writes this library with its types and operators using the BuildIt framework and BuildIt `static_var<T>` and `dyn_var<T>` types. Next, the end-user uses this abstraction to create a program by calling the operations from the library one after the other. At this point, the *static* stage of the program is executed to specialize the implementation of the library operators for the specific sequence of calls and overall program structure, or even user-specified scheduling inputs, but without any actual data. Naturally, this

requires all the operations and inputs to the library that touch the actual data in the arrays to be declared `dyn_var<T>`. The specialized and optimized program is then compiled and executed with the actual program data.

5.1.1 The BArray DSL

Before I describe the actual optimizations, let us look at the specifics of the BArray DSL and some example programs that can be written with it. The main purpose of the BArray DSL is to provide an abstraction for operating on multi-dimensional arrays of scalars (`float`, `double`, `int`) while generating optimized low-level implementations that have loop nests and scalar operations. At the heart of the DSL is the `barray<T>` template that can be wrapped around any scalar type and can hold multi-dimensional arrays of known dimensions and sizes (known at *static* time). The library then provides functions and overloaded operators to perform pointwise addition, subtraction, multiplication, and division of the arrays with other arrays and constants. The DSL also has function calls performing outer products and other operations. Figure 5.1 shows a simple program with the BArray DSL abstraction that the end-user would write. The entire program that is a sequence of calls to the individual operations in the DSL is shown in the function `test_program`, which accepts four parameters. Three of them are the dimensions of the arrays. Since in this DSL, the dimensions and sizes of the arrays are known at the *static* stage, these are declared as regular `int`s. The fourth parameter is a pointer to an array of integers, which is supposed to be the underlying buffer for one of the arrays we will operate on. The program declares three `barray<int>` `y`, `z` and `w`. Since `arr1` is passed as a parameter to the constructor of `y`, `y` is "bound" to the passed-in buffer, and operations being performed on `y` will happen on the elements of `arr1`. `z` and `w` do not have any buffers passed, and hence they just allocate their own. The sizes of these arrays are provided as arguments to the constructor as a vector of integers that also tells it the dimensions of the arrays.
Next, we see an operation `z = 0`. This is one of the first collective operations that initializes all the elements of `z` to be `0`. Similarly, `w = 1` initializes all elements of `w` to be `1`. Finally, the last statement shows the operations `y = z + w * 2` that scales each element of `w` by `2` and performs point-wise addition of the result with `z` to store the final result in `y`.
A library implementation of such an abstraction is pretty straightforward, which defines the class template `barray` and its constructor and overloads a bunch of operations on it. However, as we will see, there are a bunch of inefficiencies in a naive library implementation that can be optimized using BuildIt.

We start by looking at the basic definition of the `barray<T>` class template and its members. Figure 5.2 shows part of the `barray` template that the domain-expert developer implements. It has two members - a `dyn_var<T*>` `m_buffer` that at runtime holds the actual values to compute on and a second `std::vector<int>` `m_sizes` to hold the sizes of each dimension at compile time (*static* stage). Here, a flat buffer is used to hold the values since the dimensions of the arrays are not known at the time of writing this code. Since the sizes don't change once set, we declare the sizes as a simple `std::vector` and don't use `static_var<T>`. The figure also shows the definition of a constructor that just accepts the size and allocates a new buffer by multiplying the sizes of all dimensions.

```
#include "barray.h"

// BArray Program to be compiled
void test_program(builder::dyn_var<int*> arr1, int dim1, int dim2, int dim3) {
    // Declare 3 arrays, with 3 dimensions
    // The first arrays used a buffer passed in
    barray::barray<int> y(arr1, {dim1, dim2, dim3});
    barray::barray<int> z({dim1, dim2, dim3});
    barray::barray<int> w({dim1, dim2, dim3});

    // Initialize Z to all zeroes
    z = 0;

    // Initialize W to all ones
    w = 1;

    // Compute Z + W * 2 and store it in Y
    y = z + w * 2;
}

int main(int argc, char* argv[]) {
    // Invoke the pipeline to call BuildIt to generate code
    barray::generate_barray_program(test_program, "test", std::cout, 32, 16, 8);
    return 0;
}
```

Figure 5.1: An example program in the BArray DSL showing operations on 3-dimensional arrays

5.2 Simplifying Operator Implementation

The first optimization I will describe is operator **simplification**. Generally, this involves replacing operations in the single-stage equivalent program with simpler or fewer operations by partially evaluating some operations in the *static* stage. Such a simplification is seen in the constructor above when calculating the total size of the array. The implementation defines a function called `total_size` that iterates through the sizes of all the dimensions and multiplies them to get a single constant size. Typically, in a single-stage equivalent library for DSL, this operation would have to be performed at runtime, incurring iteration and multiplication overhead. Since in this case, the sizes are all known at the *static* stage, the generated code simply has a call to `malloc` with a fixed constant as a parameter. We call this optimization simplification, which is one of the most common optimizations when working with multi-staging. Another example of simplification is seen when implementing a function (`operator []`) to get the specific value of the array at a given coordinate at *dynamic* stage. Figure 5.3 shows the overloaded `operator[]` in the `barray` class that accepts an array of indices of type `dyn_var<int>` and indexes into the buffer after obtaining a flat index. The `builder::array` type is just a utility type that BuildIt provides that is capable of holding `dyn_var<T>` values. Since the dimension of the array is not known while writing this operation, a recursive function is required to compute the flat index. However, the generated code simply has the flat index computation inlined as a sequence of multiplications free from the overhead of recursive calls or expensive `std::vector`

```
// In barray.h
namespace barray {

template <typename T>
class barray {
    dyn_var<T*> m_buffer; // dynamic variable to hold the underlying buffer
    std::vector<int> m_sizes; // static variable to hold the sizes (and dimension) of the array
    ...
    // Multiply sizes of all dimensions to get the total size
    int total_size() {
        static_var<int> size = 1;
        for (auto x: m_sizes) size *= x;
        return size;
    }
public:
    barray(std::vector<int> sizes): m_sizes(sizes) {
        m_buffer = runtime::malloc(total_size() * sizeof(T));
    }
};

}
```

Figure 5.2: Definition of the `barray` class with a `dyn_var<T*>` for the buffer and a vector holing the *static* sizes.

lookups.

5.3 Specializing Operator Implementation

The next optimization I will describe is operator **specialization**. Specialization can be thought of as the opposite of **simplification**, where specialized code is generated for a specific program input. Specialization can lead to more code than what is present in the original implementation obtained by repeating and specializing operations for given inputs. The unrolling of loops in the `power` function we saw before is an example of specialization of the body for the exponent. However, in more complex cases, specialization can even produce more control-flow than what is present in the original code. To see an example of specialization in BArray, let us look at one of the simplest point-wise operations assignments where one `barray` is assigned to another by overloading the `operator=`. Now, generally performing any point-wise operation on an n-dimensional array would require an n-dimensional loop. But once again, since the number of dimensions is not known at the time of writing the library, a recursive function is required. As shown in Figure 5.4, the function `induce_assign_loop` iterates through a dimension and calls itself recursively for the next dimension. Calling the assignment operator on two arrays shows the generated code, which has a three-dimensional loop generated from the recursive calls. In a single-stage program, the recursive call would be extremely expensive, since the recursive calls are in a loop, meaning a separate function call is incurred for each element of the array. Furthermore, since the recursion depth depends on a parameter, general-purpose compilers are not able to optimize this automatically. A

```
namespace barray {
template <typename T>
class barray {
...
    dyn_var<int> get_flat_index(const builder::array<dyn_var<int>> &indices, int index = -1) {
        if (index == -1) index = indices.size() - 1;
        if (index == 0) return indices[index];
        else return indices[index] + (int) m_sizes[index] *
            get_flat_index(indices, index - 1);
    }

    dyn_var<T> operator[] (builder::array<dyn_var<int>> indices) {
        return m_buffer[get_flat_index(indices)];
    }
};
}
...

barray<int> x({2, 3, 4});
dyn_var<int> y = x[{1, 2, 3}];
...
// Generated code
int* var0;
var0 = malloc(96ll);
int var4 = 1;
int var5 = 2 + (3 * var4);
int var6 = 3 + (4 * var5);
int var7 = var0[var6];
```

Figure 5.3: Implementation of the `operator[]` in the `barray<T>` type. The implementation uses a recursive function to compute a flat index. The generated code just has a bunch of multiplications.

triply nested loop is a more complex control flow, but it is much faster in execution. Furthermore, our original implementation never had a triply (or any nested) loop. But the generated code has this loop nest specialized to the dimension of the array.

5.4 Fusing Operator Implementation

So far, we have looked at individual operations involving one or two `barray` variables. These operations are pretty straightforward because they do not produce any intermediates or require iterating through the range of indices multiple times. Let's now look at a more complex expression `y = z + w * 2`. This operation has 3 separate operator calls. First, the sub-expression `w * 2` is evaluated to scale each element of `w` by `2` and to create a new intermediate `barray<T>`, let's call it `m1`. The next sub-expression is `z + m1`, which performs pointwise addition of elements of `z` and the temporary `m1` to produce another temporary `m2`. Finally, `m2` is assigned to `y`, copying over all the elements. Since library implementations don't see the whole sequence of operations, the easiest but native approach for implementing this would lead to two allocations, two deallocations, and iterating

```
namespace barray {
template <typename T>
class barray {
...
    void induce_assign_loop(std::vector<dyn_var<int>*> indices, barray& other) {
        // Get the current dimension from the size of the vector of indices so far,
        unsigned index = indices.size();
        if (indices.size() == m_sizes.size()) {
            // If we are done with all dimensions just assign
            m_buffer[get_flat_index(indices)] = other.m_buffer[get_flat_index(indices)];
            return;
        }
        // Push the address of the index variable into the vector
        builder::dyn_var<int> i = 0;
        indices.push_back(i.addr());
        for (; i < m_sizes[index]; i++) {
            induce_assign_loop(indices, other);
        }
    }
    void operator=(barray& other) {
        induce_assign_loop({}, other);
    }
};
}
...
barray<int> x({2, 3, 4});
barray<int> z({2, 3, 4});
x = z;
...
// Generates
for (int var2 = 0; var2 < 2; var2 = var2 + 1) {
    for (int var3 = 0; var3 < 3; var3 = var3 + 1) {
        for (int var4 = 0; var4 < 4; var4 = var4 + 1) {
            int var7 = var4 + (4 * (var3 + (3 * var2)));
            var0[var7] = var1[var7];
        }
    }
}
```

Figure 5.4: Implemenation of the assignment operator on **barray** and the code it generates

through the range of the arrays at least 3 times. This approach is inefficient for a variety of reasons. Not only are we doing more work, we are also iterating through a large range and touching a lot of memory over and over, which affects the memory locality of this computation, severely degrading the performance on most modern architectures.

Ideally, we could iterate through the range of the dimensions exactly once, compute the expression $z_i + w_i * 2$ at each index i (i is a multi-dimensional index) and store it in y_i, all within the same loop nest. This would also not require any allocations or de-allocations. Since libraries don't have a global view of the user-written program, one common approach is to provide coarser-level hand-fused and

hand-optimized functions. In this case, the BArray DSL library could provide a function named `x_plus_cy` which, as the name signifies, takes two arrays `x` and `y` and a scalar constant `c` and returns the new array `x + c * y`. To eliminate even the last temporary and copying over, the library could further change the function to also accept a reference to the destination `barray<T>` and directly write the result into its buffer. While this approach solves the problem and is widely prevalent in high-performance libraries like cuBLAS [46]. This approach quickly stops scaling as explained in Chapter 1 since a separate function would have to be hand-written and hand-optimized for every possible expression that can appear on the right-hand side. Library writers end up implementing just the most popular kernels, leaving the less common cases to be handled by slow multiple-step operations. This leads to an inherent tradeoff between generality and performance.

The other approach that some libraries like PyTorch, TensorFlow, or Halide take is lazy evaluation. Instead of early evaluating each operation like `w * 2` or `z + m1`, the overloaded operators simply track the information about the operations being performed, carrying around references to variables used. When the value of the whole expression is required to be materialized, for instance, in an assignment, the entire operation is performed in a fused way. This is promising since arbitrary expressions on the right-hand side can be implemented in a fused way without having to handwrite each implementation. Figure 5.5 and Figure 5.6 show how lazy evaluation would be implemented in a hypothetical one-staged BArray library to solve this problem. We define an abstract class `barray_expr` that can hold arbitrary expressions of type `barray`. We then define several derived classes like `barray_constant` for representing scalar constants that appear on the RHS, `barray_sum` and `barray_product` for representing the sum and product of two other `barray_expr` and `barray_var`, which just holds a reference to a `barray<T>` being used in an RHS. The operators on these types are overloaded to recursively build the tree on the RHS. When this tree is finally assigned to a variable of type `barray<T>`, the entire loop nest is induced and the RHS is evaluated at each point by recursively evaluating each node in the tree. To facilitate this, `barray_expr` provides a virtual function `get_value_at` that accepts a multi-dimensional index and returns the value at the location for the expression. Each of the derived classes implements this virtual function by recursively calling the `get_value_at` functions from the sub-tree under them.

In our example, as the RHS is evaluated, at least 3 recursive virtual function calls are performed for each point in the range of the arrays. Even though this solves the problem of not requiring iteration multiple times or allocating temporaries, the performance is still pretty bad due to the recursive evaluation of the RHS at each point. However, we can observe that the recursive function calls and the entire tree are dependent only on the structure of the expression on the RHS and not dependent on any *dynamic* input. Thus, when we combine such lazy evaluation with BuildIt's multi-staging, the whole tree and the virtual dispatches are all evaluated, and the specialized code for the expression on the RHS is generated. Figure 5.7 shows the generated code when lazy-evaluation is applied to our multi-stage BArray implementation for the input program `y = z + w * 2`. Notice that the generated code has no recursions, no virtual calls, or not even the construction of the RHS tree. All of it has been evaluated in the first stage to produce a neat set of multiplications and sums.

5.5 Analyzing User-Programs

So far, I have described how to simplify, specialize, and fuse operations in the BArray DSL to produce neatly optimized implementations for individual statements into a single loop nest. The

```
namespace barray {
template <typename T>
struct barray_expr {
    // hypothetical singles-staged barray library
    virtual T get_value_at(std::vector<int*> index) = 0;
};

// class to hold expressions that are just a scalar constant
template <typename T>
struct barray_const: barray_expr<T> {
    T val;
    T get_value_at(std::vector<int*> index) override {
        return val;
    }
};

// class to hold expressions that are references to a variable
template <typename T>
struct barray_var: barray_expr<T> {
    barray<T>* var;
    T get_value_at(std::vector<int*> index) override {
        return var->m_buffer[var->get_flat_index(index)];
    }
};

// class to hold expressions that are the sum of two other expressions
template <typename T>
struct barray_sum: barray_expr<T> {
    barray_expr<T>* expr1;
    barray_epxr<T>* expr2;
    // Recursively get value at a given index
    T get_value_at(std::vector<int*> index) override {
        return expr1->get_value_at(index) + expr2->get_value_at(index);
    }
};
}
```

Figure 5.5: Implementation of the `barray_expr` and derived classes to lazily evaluate expressions appearing in the RHS in a hypothetical single-staged BArray library.

general idea is to use *static* variables to evaluate parts of the computations that are dependent only on the program structure and specialize the implementation for a given user-written program. However, there are still more inefficiencies in our implementation. Let's look at an example program in the BArray- `w = 1; z = w + 2;`. This program has two statements. In the first statement, all elements of `w` are set to `1`, and in the second statement, the result of the point sum of `w` and the scalar constant `2` is stored in `z`. The second operation would be optimized into a copy-and-allocation-free implementation using the fusion optimization described above. However, looking at the full program, we know that all elements of `w` must be `1` since it was set in the statement before. We should be able to eliminate both the write to `w` and the read from `w` and directly replace the value `1` in our

```
namespace barray {
// wrappers to define operator overloading
template <typename T>
struct barray_expr_t {
    barray_expr* expr;
    // Constructor from T
    barray_expr_t (const T& v) {
        auto e = new barray_const<T>();
        e->val = v;
        expr = e;
    }
    barray_expr_t(barray<T>& v) {
        auto e = new barray_var<T>();
        e->var = &v;
        expr = e;
    }
    barray_expr_t operator + (const barray_expr_t& o) {
        barray_expr ret;
        auto e = new barray_sum<T>();
        e->expr1 = expr;
        e->expr2 = o.expr;
        ret.expr = e;
        return ret;
    }
};
template <typename T>
struct barray {
...
    // induce loop that accepts a barray_expr_t appearing on the RHS
    void induce_assign_loop(std::vector<int*> indices, const barray_expr_t& other) {
        // Get the current dimension from the size of the vector of indices so far,
        unsigned index = indices.size();
        if (indices.size() == m_sizes.size()) {
            // If we are done with all dimensions just assign
            m_buffer[get_flat_index(indices)] = other.expr->get_value_at(indices);
            return;
        }
        ...
        // Recursion
    }
};
}
```

Figure 5.6: Implementation of the overloaded operators to construct the expression tree appearing in the RHS in a hypothetical single-staged BArray library. The call from the induce loop is also shown.

```
...
for (int var10 = 0; var10 < 2; var10 = var10 + 1) {
    for (int var11 = 0; var11 < 3; var11 = var11 + 1) {
        for (int var12 = 0; var12 < 4; var12 = var12 + 1) {
            int var23 = var4[var12 + (4 * (var11 + (3 * var10)))] +
                        (var6[var12 + (4 * (var11 + (3 * var10)))] * 2);
            int var26 = var12 + (4 * (var11 + (3 * var10)));
            arg0[var26] = var23;
        }
    }
}
...
```

Figure 5.7: Generating code from combining lazy-evalution of RHS with BuildIt's multi-staging for the BArray DSL.

expression. Lazy-evaluation fusion would not help us here because the result has to be materialized before the end of the statements. Even though it is obvious in this example that all elements of `w` must be `1`, but if there is any control-flow in the middle, it is hard to determine that trivially. Typically, in a traditional compiler for such a language, a full-fledged constant propagation pass would be required to determine where a variable can be replaced by a constant. Ideally, we don't want our DSL designers to resort to traditional compiler passes and analyses since that is outside their domain of expertise. We would want our DSL developers to get the benefits of these analyses and optimizations only by using the techniques they are familiar with and use in traditional libraries. To enable these kinds of analyses and optimizations, I will describe a methodology for implementing classic compiler analysis and transformations by combining *static* and *dynamic* variables. Specifically, I will describe how to implement data-flow analyses to track properties about program values and use them for specialization. A **data-flow analysis** is a static program analysis technique that computes information about the possible values or properties of program variables at each point in a program. It does so by modeling the propagation of facts along control-flow edges, using a lattice of possible facts, transfer functions for each program statement, and a meet operator to combine information from multiple paths until a fixed point is reached [103].

For the purpose of this discussion, we will implement a simple but extremely powerful and widely applicable data-flow analysis - constant propagation using just the `static_var<T>` and `dyn_var<T>` types from BuildIt without performing any explicit AST operations or changing the compiler. Essentially, we will split the execution of the library (and the user program) into two stages, where the abstract facts are processed and passed around as *static* values guiding the operations to be performed on the *dynamic* values when the actual data arrives. Even though constant propagation is a very common optimization performed automatically by general-purpose compilers like g++, clang++, or MSVC, they are able to perform it only for simple scalar data types. Performing this optimization for an arbitrary data structure from the domain, like BArray, still remains a daunting task. General-purpose constant propagation is also performed in an opportunistic way in the compiler and relies on heavy inlining and link-time optimizations. Performing it using a type-based multi-staging framework guarantees that the constants will always be propagated for any types.

The abstract property being tracked in constant propagation is whether a particular value is constant,

and if it is, what is the constant. The data-flow analysis domain for constant propagation can be defined as the mapping $variable \rightarrow \{\text{constant value}(V), \top(\text{not a constant}), \bot(\text{uninitialized})\}$. Figure 5.8 shows the lattice for this analysis.

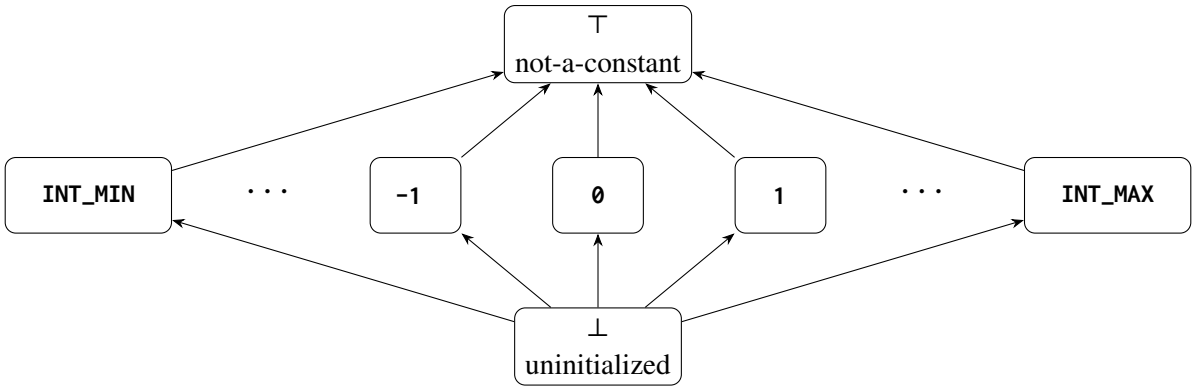


Figure 5.8: Lattice for constant propagation data-flow analysis of `int` values showing all constant values in the middle, not-a-constant at the top and an unintialized value at the bottom

For implementing an abstraction where constant `int` values can be tracked and used for optimizations, we will define a `myinteger` class library that will be exposed to the user. An integer value is being used here to simplify the example, but the same idea can be applied to our `barray<T>` class to propagate entire arrays. Figure 5.9 shows the definition of the `myinteger` class. Notice that the `myinteger` class has a `dyn_var<int>` member `value` that tracks the actual value in the *dynamic* stage. The `myinteger` class also has two `static_var<T>` members `is_constant` to track if the variable is a constant and, if it is a constant, what the value is. Notice that we do not track the bottom value since we don't care how the program behaves if a `myinteger` is uninitialized. The class also defines various constructors to initialize the value and the data-flow properties. For example, the copy constructor from a constant (not a *dynamic* value) initializes the `value` and `constant_value` to the constant and also sets the `is_constant` to `true`. The copy constructor from a `dyn_var<int>`, which is a *dynamic* value, sets `is_constant` to `false` and `constant_value` to a predetermined safe value (`0`) to avoid spurious duplication of code. The assignment operators from `int` and `dyn_var<int>` are defined in the same way. Next, we define simple binary operations like `+` to compute over the `value` and also compute the `constant_value` if both `myinteger` are currently constants. Finally, to utilize the result of the constant propagation, we will specialize in how comparisons are performed. Comparisons between two `myinteger` always return a `bool`, but if both the values are constant, the return value can be computed entirely using `constant_val`, leading to no actual condition being generated.

Figure 5.10a shows a simple program with two myinteger variables and three if-then-else statements. The `myinteger x` is initialized to a constant in the program `42` while `y` is initialized to the program input `p`. Furthermore, another `myinteger z` is initialized to the constant `0`; When this program is executed, the output shown in Figure 5.10b shows the generated output program for this program when compiled and run with the `RECOVER_VAR_NAMES=1` option for readability. First, we notice that the actual operations like `x_1 = 42`, `y_2 = arg0`, `z_3 = z_3 + 5` are all still present in the code. These are the operations on the `value` member of `myinteger`. However, the value of `x` gets constant-propagated to the condition where it is used, and it gets completely evaluated to only produce `z_3 = z_3 + 5`. At this point, the value of `z` also gets updated to `5`. On the other hand, `y` is

```
struct myinteger {
    // dyn_var to carry the actual value at runtime
    dyn_var<int> value;

    // static_vars to track properties about the value
    static_var<bool> is_constant;
    static_var<int> constant_value;

    // Various constructors to initialize the members
    myinteger(const int& c):
        constant_value(c), is_constant(true), value(c) {}

    // Set is_constant to false and constant_value to a
    // fixed dummy value so it is not left uninitialized
    myinteger(const dyn_var<int>& c):
        constant_value(0), is_constant(false), value(c) {}

    myinteger(const myinteger& other):
        is_constant(other.is_constant),
        constant_value(other.constant_value), value(other.value) {}

    myinteger(): is_constant(false), constant_value(0) {}

    // assignment operators are defined in the same way
    ...

    // Binary operations
    myinteger operator + (const myinteger& other) {
        myinteger ret = (dyn_var<int>) (value + other.value);
        if (is_constant && other.is_constant) {
            ret.is_constant = true;
            ret.constant_value = constant_value + other.constant_value;
        }
        return ret;
    }

    bool operator > (const myinteger& other) {
        if (is_constant && other.is_constant) {
            return (bool) (constant_value > other.constant_value);
        } else {
            return (bool) (value > other.value);
        }
    }
};
```

Figure 5.9: Definition of the **myinteger** class to be used by the library users. The **myinteger** class tracks the actual value as a **dyn_var<int>** and facts about whether it is a constant as **static_var<T>** variables.

```
void foo (dyn_var<int> p) {
    myinteger x = 42, y = p;
    myinteger z = 0;
    dyn_var<int> k = 0, w = 0;
    if (x > 12)
        z = z + 5;
    else
        z = z + 2;

    if (y > 13)
        w = w + 1;
    else
        w = w - 1;

    if (z > 3)
        k = k + 2;
    else
        k = k - 2;
}
```

(a) A simple program with three if-then-else statements with **myinteger** types

```
void foo_generated (int arg0) {
  int x_1 = 42;
  int y_2 = arg0;
  int z_3 = 0;
  int k_4 = 0;
  int w_5 = 0;
  z_3 = z_3 + 5;
  if (y_2 > 13) {
    w_5 = w_5 + 1;
  } else {
    w_5 = w_5 - 1;
  }
  k_4 = k_4 + 2;
}
```

(b) Output for the program in Figure 5.10a when compiled with **RECOVER_VAR_NAMES=1**.

Figure 5.10

initialized to a *dynamic* value, and the data-flow analysis result determines it as not-a-constant, and an if-then-else based on its runtime value is generated as expected with the then branch containing **w_5 = w_5 + 1** and the else branch containing **w_5 = w_5 - 1**. Since the value of **z** has been initialized with a constant and is updated through only constant operations, the value **z = 5** also gets propagated to the last condition, and it gets completely evaluated to produce **k_4 = k_4 + 2**. A high-performance library could use such types as underlying data structures for communicating results and values between successive calls to operators, and the optimizations would automatically be implemented without any runtime overhead.

In this example, we have defined the lattice (the set of values the **static_var<T>** can take), and through the overloaded operators, we have defined the transfer functions to describe how the abstract properties change through these operators. This is done in a way that can be understood and implemented by domain-expert library developers without requiring any specific compiler knowledge.

Handling Convergence

Besides defining the lattice and the transfer functions, a data-flow analysis also has to implement solving the data-flow equations till convergence. Specifically, if values affect each other circularly (in the presence of back edges), the facts for each value need to be relaxed just enough to produce a consistent fact that is true for the entirety of the program. Specifically, in this example, what happens if a **myinteger** starts out as a constant and is used inside a loop in a condition. However, before the loop body ends, the variable is now assigned from a *dynamic* value. If this is not handled

correctly, the second iteration of the loop might behave incorrectly since the specialization was done for a fact that has now changed. Before we look at what happens in this case, let us look at what the result would be if the **myinteger** is not updated at the end of the loop. Figure 5.11a shows the program with a **myinteger** used inside a condition in a loop, but the update at the end of the loop is commented out. Figure 5.11b shows the generated code with a for loop and the condition inside completely evaluated. If the update to the data-flow value is not handled correctly when the Line 14 is uncommented, the generated code would be incorrect.

```
1  void foo (dyn_var<int> p, dyn_var<int> q) {
2      // Start out z as a constant
3      myinteger z = 13;
4
5      for (dyn_var<int> c = 0; c < q; ++c) {
6          // Use the constant value of z
7          if (z > 12) {
8              runtime::puts("Then taken");
9          } else {
10             runtime::puts("Else taken");
11         }
12         // promotion of z
13         // to a dynamic value commented
14         // z = p;
15     }
16 }
```

(a) A program with a **myinteger** used in an if-then-else statement inside a loop.

```
1  void foo_generated (int arg0, int arg1) {
2    for (int var3 = 0; var3 < arg1; ++var3) {
3      puts("Then taken");
4    }
5  }
```

(b) Output for the program in Figure 5.11a when compiled with **RECOVER_VAR_NAMES=1**.

Figure 5.11

Figure 5.12a shows the program when the update is uncommented, and Figure 5.12b shows the generated code for the same. We notice that the generated code is not that straightforward. We notice that BuildIt has peeled out one iteration of the loop, which has been specialized for the constant value. The rest of the iterations are in a convoluted while loop, but have the actual condition on the **z**, as seen on Line 9. Since BuildIt inserts a backedge in the generated code only when the static tag (and all static variables) have gone to a state seen before, BuildIt notices the change to the **static_var<bool>** variable **is_constant** and does not insert a backedge. Only when it goes to **false** and stays **false** in the rest of the iteration, a loop is inserted. Thus BuildIt handles correctness automatically due to the way it handles **static_var<T>** values.

Limitations of Data-Flow Analysis Implemented using static_var<T>

We noticed above that when data-flow values change during the execution of *dynamic* loops in a program, BuildIt peels iterations to ensure correctness. The caveat, however, is that if the **static_var<T>** holding the facts of data-flow analysis changes a lot during the execution, it would lead to a blowup in the size of the code generated. To avoid such blowups, developers can ensure that with each change to a variable, the data-flow value can only move towards $\top$. In the case of

constant propagation, this can be handled by ensuring that once a variable goes to not-a-constant, it stays not-a-constant for the rest of the program, and once it takes a specific constant value, it cannot take any other constant value. If a different constant value is assigned, the variable can fallback to not-a-constant.

We also do not support the notion of a merge function where values can be moved to a less precise point in the lattice when branches merge. In fact, with this implementation, if a variable takes different analysis values on two branches, the two branches would never merge until the variable naturally converges. Even though this causes duplication of code, this is, in fact, the expected behavior. If the developer wants to avoid such duplication, they can provide a function to set the data-flow analysis to a less precise value that can be called at the end of branches manually to force merging of branches. This requires some extra effort on the part of the end-user.

Finally, the above-described methodology of packing `dyn_var<T>` variable with `static_var<T>` that tracks its data-flow properties can be used only to implement forward data-flow analyses. This method cannot be used to implement backward data-flow analyses since `static_var<T>` values naturally flow forward with the program execution. In the future, BuildIt will add support for `nd_var<T>` or non-deterministic variables, which are similar to `static_var<T>` variables, but the values can flow opposite to the program flow, which will enable backwards data-flow analyses like liveness analysis.

Even with these limitations, the methodology described is a powerful tool and can be used to implement a wide variety of optimizations in domain-specific language compilers without requiring domain experts to acquire compiler knowledge, as shown in Chapter 6.

```
1  void foo (dyn_var<int> p, dyn_var<int> q) {
2      // Start out z as a constant
3      myinteger z = 13;
4
5      for (dyn_var<int> c = 0; c < q; ++c) {
6          // Use the constant value of z
7          if (z > 12) {
8              runtime::puts("Then taken");
9          } else {
10             runtime::puts("Else taken");
11         }
12         // promotion of z
13         // to a dynamic value. z goes from
14         // constant to not-a-constant
15         z = p;
16     }
17 }
```

(a) A program with a `myinteger` used in an if-then-else statement inside a loop.

```
1  void foo_generated (int arg0, int arg1) {
2    int z_2 = 13;
3    int var3 = 0;
4    if (var3 < arg1) {
5      puts("Then taken");
6      z_2 = arg0;
7      var3 = var3 + 1;
8      while (var3 < arg1) {
9        if (z_2 > 12)
10         puts("Then taken");
11       else
12         puts("Else taken");
13       z_2 = arg0;
14       var3 = var3 + 1;
15     }
16   }
17 }
```

(b) Output for the program in Figure 5.12a when compiled with `RECOVER_VAR_NAMES=1`.

Figure 5.12

5.5.1 Applying Contant Propagation to BArray

Now that I have demonstrated a methodology to implement generic data-flow analysis in DSLs without writing any compiler-ish IR or AST manipulation code, it is easy to see how this can be applied to BArray to optimize away entire statements. Figure 5.13 shows the `static_var<T>` members added to the `barray<T>` class to track if all the elements in a variable are set to the same constant and, if they are, what the value is. Further, `operator=` is overloaded to set and update these variables. Finally, the `get_value_at` function in the `barray_var` is updated to return the constant if the whole array is the same constant. When applied repeatedly, entire programs like `w = 1; z = w + 2;` can be eliminated to produce nothing till `z` is actually materialized or till some of the elements are changed to take away the constant-ness.

```
namespace barray {
template <typename T>
class barray {
    dyn_var<T*> m_buffer;
    std::vector<int> m_sizes;
    // static_var<T> members to implement constant propagation
    static_var<bool> is_constant = false;
    static_var<T> constant_val = 0;

    void operator = (const T& v) {
        is_constant = true;
        constant_val = v;
    }
};
...
template <typename T>
struct barray_var: barray_expr<T> {
    ...
    dyn_var<T> get_value_at(std::vector<int*> index) override {
        if (var->is_constant)
            return var->constant_val;
        else
            return var->m_buffer[var->get_flat_index(index)];
    }
}
```

Figure 5.13: Implementation of constant propagation added to `barray` to track fully constant arrays.

5.6 Mapping Computations to Parallel Architectures

In the previous section, we discussed how multi-staging and domain-specific optimizations enabled by a combination of `static_var<T>` and `dyn_var<T>` variables can be used to generate efficient code to achieve both algorithmic and constant time improvements. However, a large number of applications can also exploit the performance improvements offered by parallel execution units due to the inherent parallelism present in the application, either due to domain knowledge or just general parallelization

opportunities. This parallelism could be extremely fine-grained, combining similar operations to be mapped to vector units, or at the level of long-running tasks that can be mapped to multiple threads on a CPU. In extremely parallel domains like machine learning, graph analytics, or image processing, computations can also be mapped to 10s of thousands of threads on a GPU. Modern architectures like TPUs and the newer generation of GPUs also offer application-specific parallelization units, like a matrix-multiplication unit, which can improve the efficiency of the code further. Before I explain the parallelism opportunities available in the BArray DSL, I will explain the different types of parallel hardware and ways to exploit them in existing libraries and compiler infrastructure.

Vector-level parallelism is often either directly exploited by a general-purpose optimizing compiler like gcc or clang, or the developer can manually insert intrinsics or inline assembly instructions in low-level C, C++, or Rust code. Depending on the toolchains used, the developer can also write pragmas to guide the compiler to vectorize the code better. CPU thread-level parallelism is either implemented by using a low-level threading API like pthreads or by higher-level frameworks like OpenMP [51] or CILK [150], which are implemented as extensions in C++. Finally, for GPU-level parallelism, low-level languages like CUDA [45], OpenCL [80], and Triton [178] are used. All these abstractions are either directly implemented as libraries in C and C++ or are language extensions to C++. Domain-specific application and library writers have to face the daunting task of mapping higher-level operations with parallelization opportunities to these low-level APIs. Multi-Staging specifically in a language like C++ would be an ideal choice for creating some mappings in a domain-specific way. In this section, I will explain the extensions to BuildIt to add support for mapping parallel computations to CPU threads using OpenMP and to GPUs using CUDA.

The typical compiler approach to parallelization relies on auto-parallelization techniques where the compiler performs a series of analyses to identify what parts of the computation can be performed in parallel without violating correctness and then transforms the code to map it to multiple execution units. However, in domain-specific applications and libraries, the developers are very familiar with where the parallelism is present and just need ways to write or generate code that can exploit the parallelism. For instance, for a domain like map-reduce, the developers know that during the map step, applying the mapping function to each element in the set can be done in parallel simply from the definition of the operator. They don't have to analyze the body of the function being applied. Similarly, for the reduce operation, the developers know they can perform parallel tree reductions or use atomic accesses given the semantics of the reduction. They may, however, have to make a choice depending on domain knowledge about the program or inputs to decide which kind of parallelization needs to be applied, if at all. For example, if the developer knows the number of elements is too small, the overhead of invoking map in parallel might be higher than the gains of parallelism at all. Similarly, to reduce, the developer may choose to perform tree reductions if running on CPUs, since the cost of atomics is pretty high on CPUs, or might just run the reductions entirely in parallel and use atomics when running on threads in a single block of a GPU, since atomics there are cheap. All these decisions can be made at the higher level as first-stage code, either by using analysis techniques described in Section 5.5 or with user input in the first stage. What the developers need from BuildIt are simple primitives to map the low-level operations to CPUs or GPUs.
Going back to the BArray example, we notice that there is a lot of parallelism inherent in the language. Since most of the operations that we described are point-wise, meaning they work independently on different elements in a large array, all the computations can be performed in parallel. An expert who

writes collective array operations this way is able to quickly identify the parallelism opportunities, while a general-purpose compiler for languages like C, C++, or Rust might struggle with proving that the parallel version is equivalent semantically to the serial version.

5.6.1 Mapping Computations to CPUs with OpenMP

The most popular API in C++ to map computations to threads on CPUs is OpenMP, where the developers add simple pragmas above loop nests to map iterations of loops to threads. The simplest pragma `#pragma omp parallel for` creates a parallel region and spawns a team of threads to execute blocks of code in parallel. The `for` keyword instructs OpenMP to take the next for loop and execute the iterations in parallel. These pragmas are parsed and analyzed by the C++ language frontend (gcc or clang) and lowered into calls to spawn, run, and join threads. Note that these pragmas grant the entire control to the developer and are respected by the compiler without any analysis to make sure the computations won't violate constraints. The OpenMP pragmas also have specific directives like `schedule(dynamic)` or `schedule(static)` to choose the parallelization style and load balancing between threads and directives like `reduction(+:sum)` to convert local variables to reduction variables such that operations on them are safely performed in a race-free manner. For BuildIt, we would want to follow the same model because not only is it very easy to use, but most high-performance library writers are already familiar with the APIs. Furthermore, while regular C++ pragmas are syntactically constants, we would want our pragmas to be programmable depending on the result of first-stage computations so that a high-performance DSL compiler built on top of BuildIt can make optimization decisions to generate the most optimized code.

A Generic Annotation System

Just like pragmas, we need to introduce a way to attach annotations to statements so that developers can convey parallelization and other intents and their parameters. As mentioned above, we would want these annotations to be programmable. We start by adding a function to the *builder* namespace - `void annotate(const std::string);`. This function simply accepts a string input and returns nothing. This function can be called as part of the implementation of $\mathbb{P}_S$. Any string passed to this function is attached as an annotation to the next statement generated as part of $\mathbb{Q}_{SA}$. This could be any statement, like a variable declaration, an if-then-else statement, loops, or even a simple standalone expression statement. The annotation itself does not have any effect on the execution of $eval_{static}$, and the same code is generated regardless. The annotation is just attached to the generated statement AST node from the `block` namespace. However, these annotations can be looked up later during post-processing passes before code generation. Users can write passes that can search for statements with specific annotations and transform analysis nodes. The `builder::annotate` function thus serves as a way to extend the language to provide inputs to the post-processing passes. Since the `builder::annotate` is a simple function call, unlike pragmas, it can also be called conditionally (under `static` conditions) or with the string argument constructed as a result of *static* computations. This makes the annotation system extremely flexible. Currently, each statement can be annotated with just a single annotation, but in the future, BuildIt will be extended to allow multiple independent annotations to the same statement.

Annotations for OpenMP Parallelization

Now that we have a mechanism to annotate and identify arbitrary statements in the generated code, we extend the C and C++ code generator pass to look for specific annotations. For any statement, if the annotation string begins with the prefix `"pragma: "`, the rest of the string is added as a pragma before the statement. With this extension to add an OpenMP parallel for pragma to a for loop, the developer can simply call `builder::annotate("pragma: omp parallel for")` before the for loop. The parallelization choices can also be realized as shown in Figure 5.14. Figure 5.15 shows the possible code generated based on the result of `parallel_is_profitable()` and `workload_is_irregular()`.

```
if (parallel_is_profitable()) {
    std::string pragma = "pragma: omp parallel for ";
    if (workload_is_irregular())
        pragma += "schedule(dynamic)";
    else
        pragma += "schedule(static)";
    builder::annotate(pragma);
}
for (dyn_var<int> i = 0; i < num_items; i++) {
    vec1[i] = vec2[i] + vec3[i];
}
```

Figure 5.14: Example of an OpenMP parallel for pragma added to the generated using BuildIt's annotation system.

```
// Resulting program if parallel_is_profitable() returns false
for (int i_0 = 0; i_0 < num_items_1; i_0++) {
    vec1[i_0] = vec2[i_0] + vec3[i_0];
}

// Resulting program if parallel_is_profitable() returns true
// and workload_is_irregular() returns true
#pragma omp parallel for schedule(dynamic)
for (int i_0 = 0; i_0 < num_items_1; i_0++) {
    vec1[i_0] = vec2[i_0] + vec3[i_0];
}

// Resulting program if parallel_is_profitable() returns true
// and workload_is_irregular() returns false
#pragma omp parallel for schedule(static)
for (int i_0 = 0; i_0 < num_items_1; i_0++) {
    vec1[i_0] = vec2[i_0] + vec3[i_0];
}
```

Figure 5.15: Different outputs from the program Figure 5.14 depending on the choices made in the *static* stage.

Furthermore, program-specific parameters like reduction variables can also be added to the annotation strings. Since the annotation would require the precise variable name, the variable can be given a specific name using **`with_name`** constructor helper. Figure 5.16 shows an example program and the generated code. The example shows just one variable in the reduction annotation, but the annotation can be added programmatically to include multiple variables using a **`static_var<int>`** loop. Thus, by introducing a simple primitive in BuildIt to add annotations and pragmas to the generated code, domain experts can start mapping computations to threads on CPUs in a programmable way. Other styles of CPU parallelization, like CILK, can be supported in a similar way by adding annotations to the input code, which can be handled by the C and C++ code generator to generate CILK-specific extension keywords.

```
// Create a variable declaration with a specific name. The second argument
// true instructs with_name to create a declaration for the variable
dyn_var<unsigned long long> sum_parallel = with_name("sum_parallel", true);
sum_parallel = 0;
if (parallelize_sum) {
    std::string pragma = "pragma: omp parallel for ";
    // Add variables to reduce over
    pragma += "reduction(+:sum_parallel)";
    builder::annotate(pragma);
}
for (dyn_var<int> i = 0; i < num_items; i++) {
    sum_parallel += elements[i];
}
...
// Generated code with pragma and reduction variable with specific name
unsigned long long sum_parallel;
sum_parallel = 0;
#pragma omp parallel for reduction(+:sum_parallel)
for (int i_0 = 0; i_0 < num_items_1; i_0++) {
    sum_parallel += elements_1[i_0];
}
```

Figure 5.16: Different outputs from the program Figure 5.14 depending on the choices made in the *static* stage.

Applying OpenMP Parallelism to BArray

Given this API, it is pretty straightforward to introduce CPU parallelism into the BArray DSL. We know that for point-wise operations, a lot of parallelism is available, and annotating the outermost loop with **`omp parallel for`** should give us a good improvement in performance. Applying this principle to the loop iterating over the outermost dimension of an array is also great for locality of the memory access, since different threads will access disjoint memory regions far from each other. Figure 5.17 shows the change to add the **`builder::annotate`** call before just the outermost loop by conditioning on the dimension id. To allow the end-user to exercise more control on when the parallelism is enabled, a *static* parameter as either a global variable or an explicit parameter can be used to guard the annotation.

```
namespace barray {
template <typename T>
struct barray {
...
    void induce_assign_loop(std::vector<int*> indices, const barray_expr_t& other) {
        // Get the current dimension from the size of the vector of indices so far,
        unsigned index = indices.size();
        // Base case of the recursion
        if (indices.size() == m_sizes.size()) {
            ...
        }
        ...

        // Add the OpenMP pragma if this is the outermost dimension
        if (index == 0) {
            builder::annotate("pragma: omp parallel for");
        }
        // Push the address of the index variable into the vector
        builder::dyn_var<int> i = 0;
        indices.push_back(i.addr());
        for (; i < m_sizes[index]; i++) {
            induce_assign_loop(indices, other);
        }
    }
};
}
```

Figure 5.17: Implementation to add OpenMP pragmas to the outermost loop in the BArray DSL.

5.6.2 Mapping Computations to GPUs with CUDA

Mapping parallel computations to GPUs is similar to CPUs but requires more input from the developers and work on the part of the system. For mapping computations to NVIDIA GPUs, BuildIt chooses the CUDA programming model since it is the most popular among library developers and it is sufficiently low-level to allow fine-grained control over performance. Other programming models like OpenCL and Triton can be supported in a similar way. Just like with CPUs, we want the developers to be able to control which part of the computation should run on the GPUs and how the mapping should be decided. We also want the developer to be able to decide the number of thread blocks and the number of threads per block, while the mundane tasks like outlining the GPU kernel, using the thread ids to perform the computation, should be done automatically by the system. Furthermore, we also want to make the programming interface similar to the CPU case so that computations can be moved between CPUs and GPUs with minimal changes or even a condition in the first stage. Instead of requiring the developer to write a separate function and calling it, we require the users to write the parallel computation as annotated nested for loops, similar to OpenMP. We once again leverage the generic `builder::annotate` function and support a new annotation string `"CUDA_KERNEL"` with the semantics: Any two level perfectly nested for-loop where the index is an `int` variable and the loop runs till a fixed trip count with the index variable increasing by `1` each iteration

```
dyn_var<float*> array = runtime::cuda_malloc(sizeof(float) * 32 * 512);
dyn_var<float> factor = 2.0

builder::annotate("CUDA_KERNEL");
for (dyn_var<int> outer = 0; outer < 32; outer++) {
    for (dyn_var<int> inner = 0; inner < 512; inner++) {
        dyn_var<int> tid = outer * 512 + inner;
        array[tid] = array[tid] * factor;
    }
}
```

Figure 5.18: A simple program showing a doubly nested loop in BuildIt annotated as a CUDA kernel.

is mapped to run on GPUs. We write a new CUDA Extraction Pass that runs before the C and C++ code generator. This pass looks for loop nests annotated in the above manner and performs the following transformation for each such loop nest -

1. The doubly nested loop is removed from the program, and the body of the loop is moved to a new function with an automatically generated name. This function receives the attribute **`__global__`**, to signal that this is a CUDA kernel entry point.

2. The index variable of the outer loop from the doubly nested loop is considered as the block ID, and all its occurrences in the body are replaced with **`blockIdx.x`**. Similarly, the index of the inner loop is considered as the thread ID and all its occurrences in the body are replaced with **`threadIdx.x`**.

3. Any variable defined outside the loop nest body but used inside the loop body (now the function body) is made into a function parameter, and all its occurrences in the body of the loop are replaced with the argument name. Updates to these variables are reflected back to the calling code by copying them back through the global memory. Global memory allocations and **`cudaMemcpy`** calls are inserted for each variable.

4. Any variable defined inside the loop nest and used after the loop nest is also copied back to the host using **`cudaMemcpy`**.

5. Finally, a CUDA kernel call is inserted at the point where the original loop nest was in the program, with the kernel launch parameters block size and grid size, and obtained from the bounds of the outer and inner loop, respectively. The arguments to the functions are the variables that have been referenced in the body. A **`cudaDeviceSynchronize()`** call is inserted after the kernel launch to ensure that the host code waits for the kernel to finish, since that is the semantics of the original unparallelized loop nest.

Figure 5.18 shows the input program with a doubly nested loop nest annotated as a CUDA kernel. Figure 5.19 shows the generated code with the outline **`__global__`** kernel and the host side code for calling this kernel.

```
void __global__ cuda_kernel0 (float *arg0, float arg1) {
    int tid_0 = blockIdx.x * 512 + threadIdx.x;
    arg[tid_0] = arg[tid_0] * arg1;
}
...
// Host code
float *array_0 = cuda_malloc(65536);
float factor_1 = 2.0f;
cuda_kernel0<<<32, 512>>>(array_0, factor_1);
cudaDeviceSynchronize();
```

Figure 5.19: A generated transformed program for the input program in Figure 5.18.

Thus, BuildIt supports a simple primitive to move computations to GPUs in an API that is similar to CPUs, allowing flexibility of mapping. More complex mapping strategies, including load balancing and thread and warp synchronization, can be implemented using this primitive. Barriers at different levels (grid, block, warp) can be inserted as opaque function calls recognized by the CUDA runtime. Finally, even though the transformation pass moves individual variables back and forth between kernels, the memory allocated for buffers is not moved automatically. It is up to the developer to insert calls to **cudaMalloc** and **cudaMemcpy** to allocate and transfer data for buffers as required or use CUDA's unified memory. This transformation is not done automatically since the information about the bounds of the buffers may not be available to BuildIt to compute the size of the copies. Furthermore, even if the size of the buffer is known, the entire buffer may not be accessed on the GPU, and copying it might be wasteful. Since the developer understands the access patterns, they can insert these calls. One can also build a managed buffer data type on top of these primitives that will keep track of the sizes of these buffers and insert copies automatically before and after kernel launch using `static_var<T>` variables.
Adding CUDA support to BArray is the same as the CPU case. Since for arrays with 2 more dimensions, the loop nest is already at least doubly nested, adding the `"CUDA_KERNEL"` annotation, similar to the OpenMP annotation, would move all the computation to the GPU. For single-dimensional arrays, the implementation can have a special case where the only loop is split into two loops with a fixed block size, and then the annotation is added to the outer loop.

Fusing GPU Kernel Launches

In this previous section, I described a programming model and BuildIt extensions where individual operators from the library can be mapped to a GPU to exploit the parallelism available within the operator. However, when the end-user writes a complete program with many operations, often running in a loop, each operator call will lead to a separate GPU kernel launch. GPU Kernel launches, however, have a launch overhead, and this constant overhead from launching a separate kernel for each step might end up being more than the benefits of the optimizations. Typically, in high-performance applications optimized by hand, developers launch a big kernel with individual operations as steps inside the kernel separated by synchronization of the entire grid. Even CPU parallel programming models like OpenMP or CILK maintain thread pools and merge code from different parallel loops using compiler analysis and transformations. It is imperative that we allow

```
void test_program(dyn_var<float*> arrA, dyn_var<float*> arraB, dyn_var<int> n, int dim) {
    // Square matrices A and B
    barray<float> A(arrA, {dim, dim});
    barray<float> B(arrB, {dim, dim});
    // Temporary matrix C
    barray<float> C({dim, dim});

    for (dyn_var<int> i = 0; i < n; i++) {
        C = cross(A, B);
        A = C;
    }
}
```

Figure 5.20: Example program in the BArray DSL that does repeated outer products and accumulates the result in one of the matrices.

fusing of kernels from different operator calls together without having to make massive changes to the library implementation.

Figure 5.20 shows an example from the BArray DSL with the `cross` operator, which performs the outer-product of two matrices. In this example, the outer product of the 2D matrix `B` is done with `A` a total of `n` times. In this case, a total of `2*n` separate GPU kernels are launched where `n` itself is a *dynamic* parameter. Even if, through data-flow analysis, somehow the two statements were merged, it would still take `n` separate GPU kernel launches. Since `outer_product` is not a point-wise operation, it is not possible to collapse all the products into a single operation. If the matrices themselves are small, not much work is being done in each GPU kernel launch, leading to overheads due to the constant launch cost. The performance can be greatly improved if a single GPU kernel is launched, running the for-loop itself on the GPU. Notice, however, this kind of fusion is not the same as operator fusion, where the multiple operations are combined into a single one for efficiency. The operations would still be separate, just performed as part of a single GPU launch.

Before I discuss how to fuse the kernel launches with the outer loop (and any other operations that may be part of the for-loop), let us look at how the GPU parallelized version of the cross product looks. Just like the `barray_sum` and `barray_product`, we now have the `barray_cross` class derived from the `barray_expr` class. The `get_value_at` function is interesting since it also has a loop inside. Recall that in the outer product to obtain one output value, a sum reduction is performed over the product of rows from A and columns of B. Figure 5.21 shows the implementation of the `get_value_at` function and also the annotated outer loops in the `induce_assign_loop` function. The implementation of `struct barray_expr` has been changed to add another virtual function `get_size()` that returns the output size of each sub-expression. Each assign operator call inside the loop results in a separate call to `builder::annotate("CUDA_KERNEL")`, launching a separate kernel every iteration.

We will now change this implementation for fusing. We want the end-user to be able to choose which operator calls are fused, since fusion might not always be efficient. As more operations are fused together, the kernel gets larger, stressing the register allocator. If a lot of registers are used by a single kernel, fewer threads can be launched at a time, since the register file on the Simultaneous Multiprocessor on the GPU is of a fixed size. We start by adding a function `void barray::fuse(std::function<void(void>)`. This function accepts a lambda or function pointer

```
namespace barray {
template <typename T>
struct barray_cross: barray_expr {
    barray_expr* expr1;
    barray_expr* expr2;

    dyn_var<T> get_value_at(std::vector<dyn_var<int>*> index) override {
        // Get the size of the inner dimension by looking at the sizes of one of the expressions
        int k = expr1->get_sizes()[1];
        dyn_var<T> sum = 0;
        for (dyn_var<int> i = 0; i < k; i++) {
            sum += expr1->get_value_at({index[0], i.addr()}) *
                expr2->get_value_at({i.addr(), index[1]});
        }
        return sum;
    }
};

barray_expr_t cross (const barray_expr_t& a, const barray_expr_t &b) {
    barray_expr_t ret;
    auto e = new barray_cross();
    e->expr1 = a.expr;
    e->expr2 = b.expr;
    ret.expr = e;
    return ret;
}

void barray::induce_assign_loop(std::vector<int*> indices, const barray_expr_t& other) {
    unsigned index = indices.size();
    ...
    // add CUDA kernel annotation to the outermost loop
    // for now, assume there are at least 2 dimensions
    if (index == 0) {
        builder::annotate("CUDA_KERNEL");
    }
    builder::dyn_var<int> i = 0;
    indices.push_back(i.addr());
    for (; i < m_sizes[index]; i++) {
        induce_assign_loop(indices, other);
    }
}

}
```

Figure 5.21: Implementation of the **cross** function using BuildIt's GPU annotation primitive.

and executes the whole body as a single GPU kernel. Since this should launch a single kernel, the implementation of fuse must have a doubly nested loop with the CUDA_KERNEL annotation. This fuse can then call the body inside the loop. However, the implementation of the operations must now be informed that they are already running inside a kernel and should spawn their own threads with another CUDA_KERNEL annotation. The block IDs and the thread IDs must somehow be conveyed to the implementations so that they can refer to them for mapping computations to the threads. Since there can only be one call to `barray::fuse` at a time and these calls cannot be nested, we can use global variables to communicate these values. We start by defining three global variables - `static_var<bool> on_gpu = false`, `dyn_var<int>* block_id_ptr`, `dyn_var<int*> thread_id_ptr`. The first is used to inform the operations inside the body if we are already running on the GPU and is initialized to `false`. The implementation of `barray::fuse` should set this to `true` before calling the body and set it back to `false` after. Since this information is purely dependent on program structure and no input data, it is declared `static`. Since it also mutates through the execution, it is declared as a `static_var<bool>` instead of a regular `bool`. If this is set to true, the other two variables are used to convey the block IDs and the thread IDs to the implementation of the operation. Notice that these are pointers to `dyn_var<int>` and not just `dyn_var<int>` to avoid global variables being set by threads at runtime. Pointers to `dyn_var<T>` are still *static* values. Pointers are also typically not wrapped in `static_var<T>` in BuildIt since they don't have consistent values across runs. This is okay since the mutations to these variables are not relevant to how the code is being specialized. With this implementation, each operation can first check if it is already running on the GPU. If it is, it can simply use the `block_id_ptr` and `thread_id_ptr` to perform the computation. Otherwise, it can spawn its own threads. Figure 5.22 shows the implementation of the `barray::fuse` operator, and Figure 5.23 shows the modified `induce_assign_loop` function. Here, since we are launching a bunch of operations into a single kernel, the required number of blocks and threads is not known. So the `barray::fuse` launches a number of blocks equal to the maximum number of blocks that can run on the GPU at a time and a fixed, predetermined number of threads per block. Since the actually required threads can be more than this, each block and thread does the computation of more virtual threads in the block. So there is still a doubly nested loop in the `cross` implementation. Notice, however, this loop nest is not annotated as a `"CUDA_KERNEL"`. Furthermore, since all threads need to wait till the next operation is performed to preserve semantics, a grid synchronize call is inserted at the end of each operation. This is not required in the unfused case since the end of a kernel is automatically a synchronization point for all threads. Finally, the grid sync primitive is only allowed when the kernel is launched as a cooperative kernel, which looks exactly the same implementation-wise, but has a special syntax for launch. So we modify the annotation to be `"CUDA_KERNEL_COOPERATIVE"` instead of the regular `"CUDA_KERNEL"` in `barray::fuse` to ask the BuildIt's code generator to use the cooperative launch syntax.

Figure 5.24 shows the modified program with fusion enabled around the whole for loop. The modified program shows a small change of just wrapping the whole for-loop into the `barray::fuse` call. Thus, we are able to support a complex operation like GPU Kernel fusion without having to rewrite a lot of the implementation of the operators or even having to extend BuildIt.

```
namespace barray {
// global variables to track fusion-related state
static_var<bool> on_gpu = false;
dyn_var<int>* block_id_ptr = nullptr;
dyn_var<int>* thread_id_ptr = nullptr;

void fuse (std::function<void(void)> body) {
    // Error out if the call is nested
    assert(on_gpu == false && "barray::fuse cannot be nested");

    // Spawn a GPU kernel with a fixed number of threads and blocks
    builder::annotate("CUDA_KERNEL_COOPERATIVE");
    for (dyn_var<int> block_id = 0; block_id < MAX_BLOCKS; block_id++) {
        for (dyn_var<int> thread_id = 0; thread_id < MAX_THREADS; thread_id++) {
            // Tell all operations they are already running on a GPU
            on_gpu = true;
            // capture a reference to the block_id and thread_id variables
            // Since & is overloaded to get the address at runtime, .addr() is used to
            // get the address of the actual variable
            block_id_ptr = block_id.addr();
            thread_id_ptr = thread_id.addr();

            // Now invoke the fused body
            body();

            // Unset on_gpu
            on_gpu = false;
        }
    }
}
}
```

Figure 5.22: Implementation of the **`barray::fuse`** operator to execute other operations in a fused way.

5.7 Staging Types, Data Structures and Data Layouts

So far in all Chapter 3 and the previous sections in Chapter 5, we have seen how multi-staging and BuildIt are great tools for writing general code that minimizes the effort but generates extremely specialized code that is tailored to the application. This balance of generality and high performance is achievable due to the ability to stage code (or split the execution into multiple stages). By choosing implementations of operators that are best suited for the given context, unnecessary operations can be eliminated or the right version can be chosen. However, specializing in code is only part of the process of improving performance. In a given application code, it often interacts with variables of different types, different data structures, and data layouts. As code is specialized to application needs, data structures and layouts also need to be specialized to obtain the best performance. Just like unnecessary code can be removed to remove the overhead of execution, unnecessary parts of data structures also need to be removed; otherwise, the larger memory footprint often leads to memory locality issues. In more complex cases beyond just eliminating wasteful code, combining

```
namespace barray{


void barray::induce_assign_loop(std::vector<int*> indices, const barray_expr_t& other) {
    unsigned index = indices.size();
    ...

    // if already on GPU, directly graph the block_id and the threa_id and use them
    // for the first two dimensions
    if (index == 0 && on_gpu) {
        // block * threads might be less than the outer two dimensions, add more loops
        for (dyn_var<int> py = *block_id_ptr; py < m_sizes[0]; py += MAX_BLOCKS) {
            for (dyn_var<int> px = *thread_id_ptr; py < m_sizes[1]; py += MAX_THREADS) {
                induce_assign_loop({py.addr(), px.addr()}, other);
            }
        }
        return;
    }
    // If not already on GPU, run the regular unfused implementation
    if (index == 0) {
        builder::annotate("CUDA_KERNEL");
    }
    builder::dyn_var<int> i = 0;
    indices.push_back(i.addr());
    for (; i < m_sizes[index]; i++) {
        induce_assign_loop(indices, other);
    }
}
}
```

Figure 5.23: Implementation of the modified implementation of the **induce_assign_loop** function to be support running fused.

```
void test_program(dyn_var<float*> arrA, dyn_var<float*> arraB, dyn_var<int> n, int dim) {
    // Square matrices A and B
    barray<float> A(arrA, {dim, dim});
    barray<float> B(arrB, {dim, dim});
    // Temporary matrix C
    barray<float> C({dim, dim});

    barray::fuse([&] () {
        for (dyn_var<int> i = 0; i < n; i++) {
            C = cross(A, B);
            A = C;
        }
    });
}
```

Figure 5.24: Example program showing several operator calls to **cross** on the same matrix repeatedly accumulating the result

operators can also generate more complex code. In the same way, some applications also need to programmatically choose types of variables, generate data structures and layouts to obtain the best performance. We call such customization staging types, data structure, and layouts.
We define **staging types** as a process where code in the *static* stage controls the declared types of the variables and expressions in the generated code in a programmable way. For instance, some high-performance applications can choose between varying precision of floating point operations as the precision changes with operations, as the same data moves through different parts of the application. Such optimizations are very popular in machine learning applications where activations can be stored as half-precision, full-precision, or double precision depending on choices made in the first stage code. We define **staging data structures** as a process where code in the *static* stage produces structures in the generated code in a programmatic way, adding members with customizable names, types, and ordering. At the same time, the *static* stage code also customizes how these data structures are accessed in the generated code, allowing end-to-end seamless optimizations for applications like graph-processing, networking, robotics, etc. In a similar way but not the same, we define staging data layouts as a process where the customization applies to the way bytes and bits of data are stored in specific layouts, like file systems, network packets, binary file formats, allowing precise control over bytes or bits in the layout and how they are accessed. Data layout optimizations not only improve the locality in memory by avoiding wasteful bytes and bits but can also improve the operations by optimizing the number of bit shifts and masks required.

5.7.1 Staging Types for Generic Code

Typically, in languages like C and C++, the declared types of variables are decided at the time of the variable creation and depend purely on what the variable is used for. However, by abstracting over types, developers can get maximum reuse from their code by controlling the types of variables based on context they are used in. This kind of generic code is a type of polymorphism that not only significantly reduces programmer effort but also allows parameterization of types. Such polymorphism is present in statically typed languages like C++, where functions and classes can accept template parameters that can be used to control the types and behavior of the variables. Dynamic languages also support it with runtime types where the behavior of operations depends on the type of values passed to them. Figure 5.25 shows the implementation of a simple `maxfrom` function that can return the maximum value from an array for any-sized array and types in both C++ and Python. The C++ version has a function template that accepts the type of the elements and the size of the array as template parameters and the actual array as a regular parameter. The local variables and loops refer to these template arguments. The template parameters are inferred based on the arguments passed to the function when it is invoked. Since Python is dynamically typed, no types are required, and every operation like `>` and `=` depends on the runtime type of the variable.

In BuildIt, the declared types of variables in both stages are controlled by the declared type of the variable in the `static` stage. This means to generate a variable with type `int` in $\mathbb{Q}_{SA}$, the user defines it in $\mathbb{P}$ as `dyn_var<int>`, and to generate a `struct foo`, the user writes `dyn_var<struct foo>` and so on. This means the C++ templates can be combined with the BuildIt types to write generic code that will generate $\mathbb{Q}_{SA}$ with different types. However, C++ templates need to be resolved entirely during the compilation of the *static* stage, allowing the execution of the *static* stage to exert no control on it. For example, if the user wants to generate an array in the *dynamic* stage, its type and size have to be constants in the source code of $\mathbb{P}_S$. However, many applica-

```cpp
// C++ maxfrom function using templates. Type T and the size of the array are parameters
template <typename T, size_t size>
T maxfrom(T array[size]) {
    static_assert(size > 0);
    T m = array[0];
    for (size_t i = 1; i < size; i++)
        if (array[i] > m) m = array[i];
    return m
}
...
int elems_int[5] = {...};
float elems_float[10] = {...};
maxfrom(elems_int);
maxfrom(elem_float);
```

```python
# Python maxfrom function. Types are dynamic
def maxfrom(array):
    assert(len(array) > 0)
    m = array[0]
    for i in array:
        if array[i] > m:
            m = array[i]
    return m
...
elems_int = [1, 2, 3, 4]
elems_float = [1.0, 2.0, 3.0]
maxfrom(elems_int)
maxfrom(elems_float)
```

Figure 5.25: Generic implementation of the **maxfrom** function that returns a maximum value from an array of any type and size in C++ and Python.

tions need to decide the types (or array sizes) based on some result in the computation in the first stage.

To allow this kind of generic types that depend on first-stage code, we introduce the **builder::generic** type, which is an opaque empty type that can be used as a placeholder in **dyn_var<>**. Declared variables of type **dyn_var<builder::generic>** do not have any types associated with them, and the first stage code needs to set the type before using it. We also introduce a type handle **builder::type** that can hold and manipulate types to be inserted in a generic code. BuildIt also provides utility functions to compute these types, like make an array of a given size from an element type, or get the pointee type from a pointer type, and so on. Figure 5.26 shows the implementation of the same **maxfrom** function that accepts the size as a regular parameter, which can depend on computations in the first stage. When this program is executed with parameters **elem_type = create_type<int>()** and array size say **5**, the generated code is shown in Figure 5.27. Furthermore, checks and specialization on these types can be performed as a simple if condition, as shown on Line 13, as opposed to complex and non-intuitive SFINAE checks in C++. Notice that the generated code has no reference

to the generics and produces code as if the types and sizes were specified at compile time.

```
using builder::generic;
using builder::type;
using builder::with_type;
// builder::generic does not require template parameters
dyn_var<generic> maxfrom(dyn_var<generic> array, type elem_type, size_t size) {
    // Before any variables are used, their type needs to be set
    // Create an array type with the given element type and size
    type arraytype = builder::array_of(elem_type, size);
    array.set_type(arraytype);

    // Perform type checking if necessary
    if (elem_type != create_type<int>()
        && elem_type != create_type<float>())
        assert(false && "maxfrom can only be called with float or int type");

    // Types can also be set directly using a copy constructor
    dyn_var<generic> m = with_type(elem_type, array[0]);
    for (dyn_var<size_t> i = 1; i < size; i++) {
        if (array[i] > m)
            m = array[i];
    }
    // The type for the return value needs to be set using the copy constructor
    return with_type(elem_type, m);
}

void foo (void) {
    dyn_var<int[5]> array = {0, 1, 2, 3, 4};
    size_t len = 5; // len doesn't need to be a constant
    maxfrom(array, create_type<int>(), len);
}
```

Figure 5.26: Implementation of a generic function in BuildIt where the types for `dyn_var<T>` variables are passed around and computed on like arbitrary variables

The implementation of `builder::generic` and `builder::type` is straightforward, where just like all primitive types, the `type_extractor` explained in Chapter 4 is hand-specialized to return `nullptr` for the type and a `set_type` function is added to `dyn_var<T>` to set this type later. The `builder::type` itself simply holds the AST node for types defined in the `block` namespace, and manipulation and comparison functions simply operate on this AST node inside. Finally, a new pass is inserted in the BuildIt pass pipeline to make sure no variables have their types set to `nullptr` post extraction and returns an error if it finds any. This can happen if the user accidentally doesn't set the type for a `dyn_var<generic>`. Thus, the `builder::generic` type offers an easy-to-use and programmable generic alternative to C++ templates when customizing types for the generated code in $\mathbb{Q}_{SA}$

```
void foo_generated (void) {
  int var1[5] = {0, 1, 2, 3, 4};
  int m_2 = var1[0];
  for (size_t i_3 = 1; i_3 < 5; i_3++) {
    if (var1[i_3] > m_2) {
      m_2 = var1[i_3];
    }
  }
}
```

Figure 5.27: Implementation of a generic function in BuildIt where the types for `dyn_var<T>` variables are passed around and computed on like arbitrary variables

5.7.2 Staging Data Structures

In the previous section, we discussed how `dyn_var<generic>` allows programmatically deciding the types of variables being generated for the *dynamic* stage. However, some applications require programmatically generating the user-defined types themselves depending on the computations in the first stage. Let's say we are generating a custom file system implementation based on constraints specified by the user. Such generation of a custom file system based on constraints is a good fit for implementing with BuildIt since constraints can be specified as *static* inputs during the first stage, and the implementation can generate a specialized implementation to be executed at runtime. During the generation of the filesystem, the user may specify constraints like the maximum size or whether permissions checks are required for each file. The system can then implement these features under if-then-else conditions in the first stage to only enable logic that is required. The user may also specify that the filesystem may never have concurrent read/write accesses, and expensive locking and unlocking logic can be removed from the generated code. However, custom file systems would also require custom data structures to be generated. For example, the object that holds the state of an open file handle would have permission fields if and only if permission fields are required. Similarly, the data structure holding the overall state of the filesystem would not require a lock object if concurrent accesses are not allowed. A developer could potentially add all possible members, but use only the ones required. However, not only is this approach bad for performance since it increases the memory footprint of the application, but it might also quickly explode. When programmatically generating code, an arbitrary number of fields may be required. Allocating a maximum of all features might be unfeasible.

I have already described in Chapter 3 how users can declare their own user-defined datatypes by adding `dyn_var<T>` members to structs and wrapping the structs in `dyn_var<>`. Members can be given names with the `builder::with_name` constructor, and the structs themselves can be given names using the `static type_name` member. However, in the user-defined structs, the members have to be defined inside the struct as part of the code of $\mathbb{P}_S$ itself. The only way to make it configurable would be to use templates. Not only does this make the solution ugly, but just like the section before, the choices cannot depend on the results of computations in the *static* stage. We need a way for the user to be able to describe the structs themselves programmatically in the *static* stage before creating objects of that type. Remember from Chapter 3 the only requirement for a `dyn_var<T>` to be a member of another `dyn_var<T>` is that it needs to be constructed between the constructor calls to

`member_begin<T>` and `member_end` for the parent. The constructor of the user-defined type itself runs between the `member_begin<T>` and `member` alongside its statically defined members. We can add code here to programmatically and dynamically create members. We also need a way to refer to these dynamically created members from the object.

We start by defining a Curiously Recurring Template Pattern CRTP class `dynamic_object_base<T>` in the `builder` namespace as shown in Figure 5.28. CRTP, also sometimes referred to as upside-down inheritance, is a pattern that allows defining derived class-specific functions and members in the base class itself. This is done through passing the derived class as a template parameter to the base class. What this means is that each derived class then gets its own base class, which has functions and members specific to it. This CRTP class is supposed to be used as a base class for any user-defined type that needs its members dynamically programmed. As shown in the figure, the `dynamic_object_base<T>` class defines a static member called `generators` which is a map from `std::string` to `std::function<dyn_var_base*()>`. For each programmatically created member, it creates a lambda to be executed in the constructor to create the member (recollect that `dyn_var_base` is the base class for all `dyn_var`). Since the `generator` is a static member of the class, and `dynamic_object_base<T>` is templated on the type that inherits from it, each user-defined type inheriting from this CRTP class has its own list of generators. The `dynamic_object_base<T>` class also defines a templated static member function `register_member<DT>` that creates a generator for each member that is registered. This generator, when called, heap allocates a `dyn_var<DT>`.
Next, as shown in Figure 5.28, the `dynamic_object_base<T>` defines a non-static member `mymembers` that is a map from `std::string` to `dyn_var_base*`. This map exists in every object of the user-defined type (and consequently the `dyn_var<T>` wrapped around it) and holds the actual members of that object. Finally, we just need to create all the members of this object when the constructor is called. So we define a constructor for `dynamic_object_base <T>` that iterates through all the registered members and calls the generator lambdas. The result is stored in the `mymembers` map of the object. Since the constructors for the heap-allocated `dyn_var` are now called between the constructor of the parent `member_begin<T>` and `member_end`, they are created as members inside the parent object. We also define a destructor to delete these objects when the parent `dyn_var<T>` is destroyed. Finally, we implement a `get` function that returns the requested member after derefencing the pointer. Since this function returns `builder::builder`, a copy isn't created and the same variable is returned.

Figure 5.29 shows the definition of `mytype` that uses this dynamic object interface and sets a type for the struct using the `type_name` member. The function being staged shows an object of this type that calls get to access these members by string names. The figure also shows the program using this type and the main function that registers the members before calling `extract_function_ast`. Notice that `register_member` must not be called from the function being staged since it risks being called multiple times. The main function also calls `generate_struct_decl<dyn_var<mytype>>` to generate the type definition of this custom object. The generated output in Figure 5.30 shows the code as if the two members were defined directly inside the struct and thus is free from any programmatic overhead. Specifically, even though we used `std::string` to name and identify the members, such overhead doesn't exist in the generated code.

```
namespace builder {
template <typename T>
struct dynamic_object_base {
    // Static members to store and construct generators for the type
    static std::map<std::string, std::function<dyn_var_base*()>> generators;
    template <typename DT>
    static void register_member(std::string n) {
        generators[n] = [=] () {
            return new dyn_var<DT>(with_name(n));
        };
    }

    // non-static members to hold and construct
    // the actual dyn_var<T> members
    std::map<std::string, dyn_var_base*> dyn_members;
    dynamic_object_base() {
        for (auto p: generators) {
            dyn_members[p.first] = p.second();
        }
    }
    ~dynamic_object_base() {
        for (auto p: dyn_members) {
            delete p.second;
        }
    }

    // Method to access a member by name
    builder get(std::string name) {
        return *dyn_members[name];
    }
};
template <typename T>
std::map<std::string, std::function<dyn_var_base*()>> dynamic_object_base<T>::generators;
}
```

Figure 5.28: Implementation of the **`dynamic_object_base<T>`** CRTP class.

5.7.3 Specializing Data Layouts

In the previous sections, we described extensions to BuildIt's types for programmatically controlling types of variables and a **`dynamic_object_base<T>`** implementation on top of existing BuildIt features that allows developers to programmatically create types and use them in implementations. For optimizing most high-performance applications, this gets you most of the performance improvements. However, for certain latency-critical applications where every extra byte of data stored can lead to cache misses or where the layout in memory needs to be precisely controlled to implement a binary data representation that is not left up to the compiler, this is not enough. Consider the same example of the custom filesystem described in the previous section. Besides staging the logic implementing the operations of the filesystem based on chosen features and staging the data structures kept around during the operations, like file handles and overall filesystem state, the actual

```
// Definition of the mytype user-defined struct
// that has no members but inherits from dynamic_object_base
struct mytype: builder::dynamic_object_base<mytype> {
    static constexpr const char* type_name = "struct mytype";
};
static void bar(void) {
     dyn_var<mytype> x;
     x.get("mymem") = x.get("anothermem") + 4;
}
int main(int argc, char* argv[]) {
    // Register members into mytype by calling
    // the static member function
    mytype::register_member<int>("mymem");
    mytype::register_member<float>("anothermem");

    // Extract the code and generate the struct declaration along with the code
    builder::builder_context context;
    auto ast = context.extract_function_ast(bar, "bar");
    block::c_code_generator::generate_struct_decl<dyn_var<mytype>>(std::cout, 0);
    block::c_code_generator::generate_code(ast, std::cout, 0);
    return 0;
}
```

Figure 5.29: Implementation of a user-defined type **`mytype`** and the main function registering the members.

```
struct mytype {
  float anothermem;
  int mymem;
};
void bar (void) {
  struct mytype var0;
  var0.mymem = var0.anothermem + 4;
}
```

Figure 5.30: Generated program from the output of the program in Figure 5.29 showing the generated struct and the code accessing it

data stored on the disk and its layout also depend on the features chosen. For example, if file permissions like read/write/executable are enforced, three bits of information need to be added to the inode stored on the disk. If the sizes of the files are bounded by a fixed, known size at compile time, the field storing the size of the file in the inode can be shrunk to just enough bits to represent the size. Furthermore, if the number of files in a directory is bounded, the inode structure can be simplified to just have an array of entries instead of needing an extendable list, simplifying both the layout and the code needed to access this.

Such precise control is currently not possible with the data structures in BuildIt. Even if tra-

ditional ways of implementing binary layouts using packed structs with bit fields are used to shrink the number of bits required, extra operations might be required to access the true value. For instance, imagine a file system deployment where the size of a file in a file system is guaranteed to be at most 128 bytes, but always more than 64 bytes. One way of representing this in the field in the inode would be to use 7 bits to represent the length field ($log_2(128)$. However, that would be wasteful since the range 64-128 can be represented in merely 6 bits by subtracting the minimum value 64 from the length before storing it on the disk. Now, accessing the length field would require adding 64 on a read and subtracting 64 while writing. A C struct with bit fields wouldn't do this automatically.

Other alternatives to describe such precise data layouts in memory and generating operations to access them are to use a full-fledged DSL like ProtoBuf [181]. ProtoBuf provides a language interface to specify properties about layouts like the type of fields, their ranges, indirection between fields, and so on. A specialized compiler then generates functions to read/write each field, given the base pointer of the layout. Although ProtoBuf is a powerful DSL, it requires developers to write or generate the layout description in an entirely new language. Since staging of code and data structures is happening in C++ under BuildIt, the `static` stage would also have to generate ProtoBuf input on the side to be fed as input to the ProtoBuf compiler to generate code used in the `dynamic` stage. Not only is this approach cumbersome and requires developers to learn a new language, but the options for configurability in ProtoBuf are very limited and do not offer the fine-grained control most low-latency applications need.

Layout Customization Layer

To solve this problem in a general way for all applications requiring precise programmatic control over the data layouts, we make the observation that traditional ways of implementing data layouts, i.e., structs, are just a way to describe to the C or C++ compiler how to access a particular field in memory. If we make this process of describing the layout to the compiler into explicit code, the problem of staging layouts reduces to the already solved problem of staging code. To achieve this, I will introduce a new **Layout Customization Layer** on top of BuildIt. This layer is implemented purely by using BuildIt's `dyn_var<T>` and `static_var<T>` types and does not require any extensions to the system itself. In this section, I will describe the workflow and the API to use the layout customization layer, followed by the implementation on top of BuildIt.

We use the term *Layout* to refer to a specific collection of fields with fixed sizes, ordering, and alignment that describes how the data would be laid out in a buffer and can be used to generate code to read or write the fields. Notice that *Layouts* themselves are not data or objects in memory but just an abstract way of describing the fields and their properties like C structs. We introduce a new type `dynamic_layout` to hold a *Layout* in the *static* stage, while the abstract class `dynamic_member` holds an individual field in a *Layout*. The following are the steps to define and customize a Layout:

1. The developer starts by declaring an object of type `dynamic_layout`. The object starts as Empty and the fields of type `dynamic_member` can be added along with their properties identified by a string name using the `add_member` function.

Type	Member function
`dynamic_layout`	`void add_member(std::string name, dynamic_member*, int group)`
	`void apply_policy(set<int> groups, Policy p)`
	`void finalize_layout(void)`
	`dynamic_member* operator[] (std::string name)`
	`void print_layout(std::ostream)`
`dynamic_member`	`virtual dyn_var<char*> get_addr(dyn_var<char*> buffer)`
	`virtual dyn_var<long> get_integer(dyn_var<char*> buffer)`
	`virtual void set_integer(dyn_var<char*> buffer, dyn_var<long> value)`
`generic_int_member<T>`	`generic_int_member(int flags)`
	`void set_range(T min, T max)`
	`void set_alignment(int bits)`

Table 5.1: User side API for the `dynamic_layout`, `dynamic_member` and `generic_int_member<T>` types.

2. After all fields have been added, the developer chooses an optimization strategy by calling the `apply_policy` function and finalizes the layout by calling the `finalize_layout` function. At this point, the order, sizes, and exact offset of each field have been decided.

3. Finally during the actual implementation of the functions to stage, the code can use this `dynamic_layout` object to access the fields from a buffer, again identifying them by the string names by the use of the overloaded `operator[]` and the `get_*` and `set_*` functions on the returned fields.

The final step above generates the specialized implementation for the application, interleaved with the specialized code to access the data layouts in memory. Table 5.1 shows the functions and their signatures. Notice that the `add_member` function also accepts a group for each field. Thus allowing the user to apply different policies for different groups with `apply_policy`. Since the `dynamic_member` is just an abstract class, we also show the API for the `generic_int_member<T>` derived class, which can be used to represent common integer-like fields.

Figure 5.31 shows a simple filesystem inode metadata example with `is_readable`, `is_writable`, and `size` fields added as `generic_int_members` of different types. The figure also shows some code snippets accessing these fields by calling the `operator[]` and `get_integer` and `set_integer` functions.

This configuration and access look very similar to staging data structures; however, the difference is in the generated code. The generated code doesn't have any structs or members generated, but directly the code to index into the buffer and de-reference the fields at the right address, as shown in Figure 5.32

The implementation of the type `dynamic_layout` is shown in Figure 5.33, where it simply holds different `dynamic_members` in a map and a vector of vector of strings to organize the fields in different groups. The `operator[]` simply retrieves the members. The abstract base class `dynamic_member` is also shown, which has an internal API function called `get_size()`, `get_align()`, and `get_offset()` which returns the size, alignment requirement and offset of this field in the layout in number of bits from the beginning of the layout. While the `get_size()` and `get_align()` functions are implemented by the derived classe like `generic_int_member<T>`, the `get_offset()` is always defined as adding the offset of the previous field in the order and the size of the previous field in the order and aligning it to the

```
dynamic_layout file_metadata;
file_metadata.add_member("is_readable", new generic_int_member<bool>(flags::aligned));
file_metadata.add_member("is_writable", new generic_int_member<bool>(flags::aligned));
file_metadata.add_member("size", new generic_int_member<unsigned>(flags::aligned));
...
// Inside the implementation of |ℙ_S|
// Pointer to the base of the layout
dyn_var<char*> base = ...;
file_metadata["is_writable"]->set_value(base, true);
dyn_var<unsigned> file_len = file_metadata["size"]->get_value(base);
```

Figure 5.31: Example of a file system programmatically adding fields to the **file_metadata** data layout and then accessing the fields using get layout and set layout

```
bool* var_0 = (void*)(base_1 + 1); // is_writable field is at the byte offset 1
var_0[0] = 1;
unsigned* var_2 = (void*)base_2; // size field is at the byte offset 1
unsigned len_3 = var_2[0];
```

Figure 5.32: Output of the code generated from the implementation in Figure 5.31.

alignment requirement of the current field.

The **apply_policy** function, not shown here, rearranges the fields in a set of groups to optimize a specific criterion. Currently, the only policy supported is the **OptimizeWidth** policy that minimizes the number of bits required to represent he layout. Since fields can have different alignment requirements, the total size of the layout also depends on the order in which they are placed. The current implementation of the policy takes a brute force strategy that simply tries all permutations to find the best ordering (treating fields with the same size and alignment requirements as the same). This is not too expensive for common layouts with 10-20 fields. However, a new strategy that uses a dynamic programming-based implementation has been developed by my mentee, Dhruv Saraff, in his master's thesis [153] that achieves optimal ordering in less than 2 mins for more than 30 fields. This policy is run entirely in the *static* stage, adding no runtime overhead. In the future, policies for optimizing the number of bit-shifts and bit-masking for fastest access would be added.

Next, the Figure 5.34 shows the implementation of the derived class **generic_int_member<T>** that implements the **get_offset**, **get_align**, **get_integer** and **set_integer** functions. The basic implementation of the **get_integer** and **set_integer** functions simply accesses the field by adding the offset of the field to the base of the buffer and treating the value at the location as being type **T**. While the computations on the sizes and offsets are done in the *static* stage and the name lookups and the virtual dispatches are also in the *static* stage, the actual indexing into the buffer and manipulating the value is done with **dyn_var<T>**, resulting in the code generated in the second stage. This implementation shows how the code in Figure 5.32 is free from all overheads and has the precise logic to retrieve fields from the layout.

The above shown implementation of the **generic_int_member<T>** gives us byte-optimized layouts, but the representation can be shrunk to individual bits, especially when the range of values is

```
struct dynamic_member {
    dynamic_member* prev;
    // virtual functions defined below
    virtual size_t get_size() = 0;
    virtual size_t get_align() {
        return 8; // default alignment for all fields in byte aligned
    }
    size_t get_offset() {
        return alignto(prev->get_offset() + prev->get_size(), get_align());
    }
    ...
    // Rest of the API virtual functions
};
struct dynamic_layout {
    // map and groups to hold the registered fields
    std::map<std::string, dynamic_member*> field_map;
    std::vector<std::vector<std::string>> field_groups;

    void add_member(std::string name, dynamic_member* m, int group) {
        field_map[name] = m;
        if (field_groups.length < group + 1)
            field_groups.resize(group + 1);
        field_groups[group].push_back(name);
    }
    void finalize_layout(void) {
        // The policy has already been applied, and the fields
        // have been rearranged. Just set the prev pointer
        dynamic_member* prev = nullptr;
        for (auto v: field_groups) {
            for (auto m: v) {
                field_map[m]->prev = prev;
                prev = field_map[m[];
            }
        }
    }
    // Retrieve the field requested by the application
    dynamic_member* operator[] (std::string name) {
        assert(field_map[name] != nullptr && "Requested field not found");
        return field_map[name];
    }
};
```

Figure 5.33: Implementation of the **`dynamic_member`** base class and the **`dynamic_layout`** class.

```
template <typename T>
struct generic_int_member {
    int align; // alignment can be configured
    generic_int_members(int f) {
        if (f & flags::aligned) align = alignof(T) * 8;
        else align = 1;
    }
    // Function to set custom alignment if required
    void set_align(size_t a) {
        align = a;
    }
    size_t get_size() override {
        return sizeof(T) * 8; // Basic implementation returns the typical size
    }
    size_t get_align() override {
        return align;
    }
    dyn_var<long> get_integer(dyn_var<char*> buffer) {
        assert(get_offset() % 8 == 0);
        dyn_var<T*> ptr = (T*)(buffer + get_offset() / 8);
        return *ptr;
    }
    void set_integer(dyn_var<char*> buffer, dyn_var<long> v) {
        assert(get_offset() % 8 == 0);
        dyn_var<T*> ptr = (T*)(buffer + get_offset() / 8);
        *ptr = (T)v;
    }

};
```

Figure 5.34: Implementation of the **generic_int_member<T>** class and its member functions.

restricted. Figure 5.35 shows the optimized implementation, which adds the **set_range** function that accepts a range of possible values for the field and computes the exact number of bits required. The **get_integer** and **set_integer** functions are similarly modified to perform bit manipulations to compute the exact value. Notice, even though the **get_integer** looks complex, a lot of the computation is on **static_var<T>** A values and thus is completely evaluated to generate the most efficient bit masking and shifts to obtain the value.

Finally, Figure 5.36 shows an implementation where the layout is optimized to just use one bit for **is_readable** and **is_writable**, and the range for **size** is set to 64-128. All the fields are also marked to be unaligned. Thus, packing the entire layout into just a single byte. Besides changing the layout description, the application code is exactly the same. The generated code after this change is shown in Figure 5.37. Notice that the code now uses bit manipulations to access the values, but is still free from any overhead.

Thus, we have implemented a very generic Layout Customization Layer on top of BuildIt that offers applications the ability to stage layout and control the bit-level layouts programmatically, improving both memory access latencies and operations done with the data.

```
template <typename T>
struct generic_int_member {
    ...
    T range_begin = 0;
    // Default number of bits is the same as before
    size_t num_bits = sizeof(T) * 8;

    void set_range(T rb, T re) {
        range_begin = rb;
        num_bits = bits_required(rb - re);
    }
    size_t get_size() override {
        return num_bits;
    }
    dyn_var<long> get_integer(dyn_var<char*> base) {
        builder::dyn_var<long> value;
        int bit_start = get_offset();
        int bit_end = bit_start + get_size();
        if (bit_start % 8 != 0) {
            int lbs = bit_start;
            int lb = byte_size - lbs % byte_size;
            if (lb > m_bit_size)
                lb = m_bit_size;
            int lbe = lbs + lb;
            int ubs = lbe;
            int ube = bit_end;
            if (ube - ubs != 0)
                value = extract_bits_lower(base, lbs, lbe)
                    | extract_bits_upper(base, ubs, ube) << lb;
            else
                value = extract_bits_lower(base, lbs, lbe);
        } else {
            value = extract_bits_upper(base, bit_start, bit_end);
        }
        if (range_begin != 0)
            return value + range_begin;
        else
            return value;
    }
};
```

Figure 5.35: Optimized implementation of **`generic_int_member<T>`** that handles bit manipulations.

```
dynamic_layout file_metadata;
auto is_readable = new generic_int_member<bool>(0); // 0 implies field is not aligned
is_readable->set_range(0, 2); // Set the range to allow only 0/1
file_metadata.add_member("is_readable", is_readable);
auto is_writable = new generic_int_member<bool>(0); // 0 implies field is not aligned
is_writable->set_range(0, 2); // Set the range to allow only 0/1
file_metadata.add_member("is_writable", is_writable);
auto size = new generic_int_member<unsigned>(0); // 0 implies field is not aligned
size.set_range(64, 128); // Set the range to 64-128 bytes
file_metadata->add_member("is_writable", size);
...
// Implementation of |ℙ_S| stays the same
dyn_var<char*> base = ...;
file_metadata["is_writable"]->set_value(base, true);
dyn_var<unsigned> file_len = file_metadata["size"]->get_value(base);
```

Figure 5.36: Filesystem implementation using the optimized version of **`generic_int_member<T>`** and calling **`set_range`** to pack the fields efficiently

```
char var_0 = 1 << 1; // The is_writable is at the bit location 1
base_1[0] = (base_1[0] & 0b11111101) | var_0; // Erase the old value and write the new
// size field is the upper 6 bits of the byte, 64 is added to the result
unsigned len_2 = base_1[0] >> 2 + 64;
```

Figure 5.37: Output of the code generated from the implementation in Figure 5.31.

5.8 Hoisting Conditions with "The Trick"

So far, I have described several techniques to optimize, specialize, and simplify based on **`static_var<T>`**. However, sometimes applications need to specialize the implementation based on values known only during the *dynamic* stage. This might be required for multiple reasons, including indexing into data structures not available at the *dynamic* stage or resolving fine-grained branches depending on *dynamic* parameters or just calling a function that cannot be called at the *dynamic* stage. Unfortunately, *dynamic* values cannot be converted to *static* values directly. To be able to do this, I introduce a function written as a purely library on top of BuildIt shown in Figure 5.38. From the signature, the function accepts a **`dyn_var<T> &`** and a range of **`T`** and returns a **`static_var<T>`**. It does so by iterating through the range with a *static* loop, comparing the iterator with the passed value, and returning it if it matches. None of the operations in this function is illegal, and the implementation logically returns the equal value, but as *static*. The trick is that it branches over all possible values of **`v`** and returns the right one. Since it returns a **`static_var<T>`** which is live after returning and has a different value with each iteration of the loop, the branches do not merge and all the code after the return from the function is moved inside the branch automatically, Inside the branch however, the value is accessible as a concrete **`static_var<T>`** and can be used to do anything that can be done with a **`static_var<T>`** like index into the array **`arr`** that is only available in the *static* stage as shown in the example.

```
template <typename T>
static_var<T> up_cast_range(dyn_var<T> &v, T range) {
  static_var<T> s;
  for (s = 0; s < range - 1; s++) {
    if (v == s) {
      return s;
    }
  }
  return s;
}
...
dyn_var<int> bar(dyn_var<int> x, dyn_var<int> y) {
    const int arr[5] = {21, 13, 11, 112, 9};
    static_var<int> xs = up_cast_range(x, 5);
    if (xs % 2 == 0)
        return arr[xs] + y;
    else
        return arr[xs] - y;
}
```

Figure 5.38: An **up_cast_range** function written on top of BuildIt that can convert **dyn_var<T>** to **static_var<T>** given a range of possible values.

The generated code when running this is shown in Figure 5.39. As we can see, the generated code is specialized and simplified since the indexing into the *static* array is resolved. The condition based on **xs** has also been completely evaluated. If the **up_cast_range** wasn't used, the condition would be generated as it is. This technique thus harvests conditions on **dyn_var<T>** values into a single condition at the top, allowing all smaller conditions to be completely resolved in **static** stage, thus effectively allowing specialization based on *dynamic* values. This technique is simply referred to as "The Trick" in most multi-staging, literally and often requires frameworks to manually implement it. In the case of BuildIt and **REMS** due to correct handling of side-effects, this can be implemented just as a library function in a few lines. Other similar functions can also be implemented to accept different ranges of values as potential options, including a set of values or a generator, and so on.

```
int bar_specialized (int arg0, int arg1) {
  if (arg0 == 0) {
    return 21 + arg1;
  }
  if (arg0 == 1) {
    return 13 - arg1;
  }
  if (arg0 == 2) {
    return 11 + arg1;
  }
  if (arg0 == 3) {
    return 112 - arg1;
  }
  return 9 + arg1;
}
```

Figure 5.39: Output of the program in Figure 5.38 with a giant if-then ladder with each branch specialized for different concrete values of the variable `x`.

Chapter 6

Case Studies for Developing DSLs with BuildIt

> "The function of good software is to make the complex appear to be simple"
>
> Grady Booch

In Chapter 3 I introduced the BuildIt multi-staging framework that implements **REMS** in C++ in a lightweight way. We discussed the design decisions and programming model of the framework and why implementing it in C++ is the ideal choice for implementing high-performance DSLs. In Chapter 4 we looked at the actual implementation of BuildIt. Specifically, we looked at how BuildIt implements each part of the evaluation procedure $eval_{static}$ as described in **REMS**, especially implementing repeated execution to extract all control-flow in the program. In Chapter 5, we discussed what features are required to implement high-performance programming languages and libraries with a system like BuildIt and discussed implementation of these features as either extensions to BuildIt or implemented them on top of the BuildIt types. In this Chapter I will discuss real world applications of the techniques to implement a high-performance DSLs for the graph applications running on GPUs, for implementing a compiler for generating ad-hoc networking protocols given features and deployment constraints suitable for running in datacenters and edge devices and for turning an interpreter for regular expressions into a compiler by the use of Futamura Projections [69] that generates code on par with state-of-the-art regex engines. All these case studies will focus on minimizing the effort required to implement the DSL compiler while writing as little compiler-ish code as possible.

6.1 EasyGraphit: A GraphIt to GPU Compiler Using BuildIt

In Chapter 1, we discussed that to improve the productivity of both domain-expert developers and end-users, it is essential that we focus on techniques that allow us to write library-like code while still generating code that is on par with code generated from full-fledged compilers. One of the very popular high-performance domains is graph processing, where users run applications like Breadth First Search (BFS), Shortest Path (SSSP) on graphs with billions of edges and millions of vertices. The reason this domain is so interesting is that a lot of high-performance algorithms from big data, fraud detection, navigation, and weather simulations can be reduced to graph processing. Generating good performance code for these applications is the key to powering a variety of domains.

Furthermore, a lot of high-performance libraries and DSL compilers have been built for this domain, motivating a bunch of optimizations and novel code generation strategies. Libraries like Ligra, Julienne, GAPBS, GSwitch, Gunrock, and SepGraph have been written to target graph applications on CPUs and GPUs. Furthermore, compiler techniques like GraphIt [201, 25, 23, 198] have also been developed that can generate state-of-the-art code for running on CPUs, GPUs, and other specialized hardware like Hammerblade and Swarm [95]. In this section, I will describe the design of EasyGraphit [20], a compiler written using BuildIt that applies the techniques and extensions introduced in Chapter 5 to the graph domain implemented in as few as 2021 lines of code while generating state-of-the-art code for running on GPUs.

6.1.1 The Graph Domain Abstraction

Before we discuss how the graph application compiler is implemented with BuildIt, I will explain the abstraction we will use to write these applications. We will build the compiler using the abstraction introduced by the GraphIt DSL. The GraphIt DSL uses a bulk synchronous model where sets of vertices and edges are processed by applying functions on them repeatedly till a convergence condition is met. We choose this model since the GraphIt DSL has shown to be a good fit for these applications when targeting massively parallel architectures like CPUs and GPUs [201, 198, 25]. The GraphIt DSL takes the approach of user-provided scheduling to tailor the generated code to variations in data distribution. I will discuss an example program and how it is compiled differently for different types of graphs. The GraphIt compilers take two programs - an algorithm input that specifies what to compute and a scheduling input that specifies how to compile the code, i.e., what optimizations to apply. This abstraction gives control to the user so that they can rapidly explore the optimization space to find the best compile program. The GraphIt DSL also implements a notion of Hybrid Scheduling where multiple schedules can be combined to be chosen at runtime based on the state of the execution. The GraphIt compiler implements this using a traditional compiler design technique with its own parser, IR, analyses, and transformations, and target-dependent code generation.

```
1 EdgeSet edges; VertexSet active_set;
2 ...
3 edges.from(active_set).to(not_visited).apply(updateEdge);
4 ...
5 func updateEdge(Vertex src, Vertex dst)
6     new_ranks[dst] += old_ranks[src];
7 end
```

Figure 6.1: A small program written in the GraphIt DSL highlighting the GraphIt abstraction - types and operators.

For this case study, we will use the same model so we can compare the performance and the effort required to implement the DSL in a fair way. Figure 6.1 shows a part of a GraphIt program highlighting the domain-specific abstraction - types and operators. The first line shows the data

types in the GraphIt language. The `EdgeSet` is a data type for holding a set of edges to be processed, while the `VertexSet` is similarly a data type for holding a set of vertices to be processed. The next line shows the second part of the abstraction - the operations. This line can be broken down into individual operations as follows:

1. `edges.from(active_set)...`: start with all the edges in the set `edges`. From this set filter and keep the edges that originate from the `active_set`, meaning the edges where the source vertex lies in the `active_set`.

2. `...to(not_visited)`: from the remaining edges, further filter and keep the edges where the destination vertex satisfies the `not_visited` property. The `not_visited` is a user-defined function (UDF) not shown here that checks if a vertex has already been visited.

3. `...apply(updateEdge)`: on the remaining edges apply the user-defined function `updateEdge`. The `updateEdge` UDF shown below accepts two vertices, the source and destination of the edge, and performs the computation `new_ranks[dst] += old_ranks[src];`.

We can identify this application as a modified version of the popular PageRank application, where only a subset of edges is processed each round. These are the operations available in GraphIt's bulk synchronous model. As we can see, the operators are very abstract and do not specify anything about how the apply functions or the filter functions should be implemented. Specifically, it doesn't specify the order in which the edges should be processed, whether they can be invoked in parallel, or even what low-level data structure should be used to implement these operations.
If a graph domain-expert who has written a lot of high-performance graph applications were to implement this specific sequence of operations, they would present three options shown in Figure 6.2a, Figure 6.2c, and Figure 6.2d. The first implementation is the most optimum version when the vertex set `active_set` is sparse, meaning a small fraction of all the vertices in the graph are active. This is ideal because we iterate through only the active vertices in parallel and their neighbors serially, check the destination property, and apply the function. This helps eliminate a lot of edges that do not originate from `active_set`. The second implementation is the most optimum version when the vertex set `active_set` is medium dense, meaning many but not most vertices are in the `active_set`. This is better because here we iterate through all the edges in parallel and get more parallelism out of the computation. Finally, in the case the `active_set` is very dense, meaning almost all vertices are in the `active_set` the best strategy shown in the third implementation first transposes the edges (swaps the source and destinations), then iterates in parallel through the destinations, checks the destination property, then for each destination iterate through the sources and checks the source property and finally apply the `updateEdge` function. This is ideal because we eliminate a lot of destinations early, avoiding wasted work going through their sources.

Furthermore, the user-defined function `updateEdge` is also implemented differently. In the two cases, since we are in parallel iterating through the source, it is possible that two sources mapped to different threads try to update the same destination vertex property at the same time. To avoid races, an atomic access is used as shown in Figure 6.2b. Whereas in the third case, since we are iterating through destinations in parallel, we guarantee that each destination vertex is assigned to a different thread and no races are possible. To improve performance, we can eliminate the atomic access.

What I just explained above is the domain knowledge from the graph, which makes these implementation choices semantically correct. In any other domain, this might not always be true.

```
parallel for (v in active_set.vertices) {
    for (neigh in edges.neighbors(v)) {
        if (not_visited(neigh)) {
            updateEdge(v, neigh);
        }
    }
}
```

(a) Possible implementation for the program in Figure 6.1 when the **active_set** is sparse.

```
void updateEdge(int src, int dst) {
  atomicAdd(&new_ranks[dst], old_ranks[src]);
}
```

(b) Possible implementation for the UDF **updateEdge** when the **active_set** is sparse or medium dense.

```
parallel for ((src, dst) in edges) {
    if (src in active_set) {
        if (not_visited(dst)) {
            updateEdge(src, dst);
        }
    }
}
```

(c) Possible implementation for the program in Figure 6.1 when the **active_set** is medium dense.

```
parallel for (v in edges.vertices) {
    if (!not_visited(v)) continue;
    for (n in edges.transponse.neigh(v)) {
        if (n in active_set) {
            updateEdge(n, v);
        }
    }
}
```

(d) Possible implementation for the program in Figure 6.1 when the **active_set** is very dense.

```
void updateEdge(int src, int dst) {
  new_ranks[dst] += old_ranks[src];
}
```

(e) Possible implementation for the UDF **updateEdge** when the **active_set** is very dense.

We need to leverage this domain knowledge while implementing the compiler. If we ask the domain-experts to implement these abstractions, they would write an operator function for **apply** in their language of choice, like C++ or CUDA, and insert a giant if-then-else to check this property and launch the appropriate version. The implementation of this operator is shown in Figure 6.3. In fact, this is how most of the high-performance libraries are implemented. The downside of this approach is that the conditions are checked at runtime and an overhead is incurred, which could be removed if these branches were partially evaluated ahead of time and the appropriate version was generated. Notice the **updateEdege** functional implemented this way. The condition to decide which version to launch is checked on each edge, undoing any of the performance improvements from the atomics. Some of the graph datasets have billions of edges to process, and even with branch predictors, this overhead would be substantial. Naturally, a library like this would be ripe to be

converted into a 2-stage evaluation system using BuildIt's multi-stage evaluation capabilities. This is exactly how we will implement the GraphIt DSL on top of BuildIt.

```
// Conditions to check for the sparsity level of active_set
if (active_set.is_sparse) {
    parallel for ((src, dst) in edges) {
        if (src in active_set) {
            if (not_visited(dst)) {
                updateEdge(src, dst);
            }
        }
    }
} else if (active_set.is_dense && !is_large(active_set)) {
    parallel for (v in edges.vertices) {
        if (!not_visited(v)) continue;
        for (n in edges.transponse.neigh(v)) {
            if (n in active_set) {
                updateEdge(n, v);
            }
        }
    }
} else {
    parallel for (v in edges.vertices) {
        if (!not_visited(v)) continue;
        for (n in edges.transponse.neigh(v)) {
            if (n in active_set) {
                updateEdge(n, v);
            }
        }
    }
}
...
void updateEdge(int src, int dst) {
    // Condition checked for each and every edge processed.
    if (active_set.is_sparse  || active_set.is_dense && !is_large(active_set)) {
        atomicAdd(&new_ranks[dst], old_ranks[src]);
    } else {
        new_ranks[dst] += old_ranks[src];
    }
}
```

Figure 6.3: A library style implementation of the **`apply`** operator and the **`updateEdge`** UDF in GraphIt branching over various sparsity levels of the **`active_set`** and dispatching the optimal version

6.1.2 GraphIt DSL Implementation

Now that I have described the GraphIt domain abstraction and the possible optimizations under different conditions, I can describe the implementation of our compiler. In this section, I will go step-by-step through various components of our implementation and describe how it is implemented

Major Component	GraphIt LOC	EasyGraphit LOC
Frontend and Abstraction	8,944	55
Scheduling	2,240	151
Analysis and Transformations	10,358	1,320
GPU and Host Code generation	550	2,188
Total	22,092	2,021

Table 6.1: The lines of C++ code required to implement the original GraphIt compiler and EasyGraphit showing about 11x reduction in code size. The total for the EasyGraphit column above doesn't match the exact number since a few lines are counted twice in multiple components.

using the techniques described in Chapter 5. I will also take this opportunity to compare this implementation with the original GraphIt DSL compiler targeting GPUs [25] to demonstrate the savings in implementation effort. I will divide the overall implementation into the following components: 1. Frontend and abstraction, 2. Scheduling Interface and Implementation 3. Analysis and Transformations and 4. Code Generation for both host and device.

Before getting into the implementation of each component, I will clarify how BuildIt's *static* and *dynamic* stages are mapped here. Let's first identify the *static* and *dynamic* inputs. The end-user who writes the programs in the DSL provides, during the first stage, an instance of a GraphIt program along with the scheduling inputs that correspond to the optimization choices. So the structure of the program itself, i.e., which operators are called and in what order, would be available in the first stage. This is similar to the BArray language seen in Chapter 5. Thus, any analysis that takes into account only the structure of the program can be evaluated in the *static*. Furthermore, the developer would be supplying explicit optimization decisions in the first stage - like the choice of order of iteration, parallelization, data structure choices, and so on. Any specialization and simplification based on these choices would be performed in the *static* stage. All the other inputs, like the actual graphs to process, the values associated with vertices and edges, like edge weights or ranks of vertices in the program above, would be only provided when the generated program is run. So these would have to be treated as *dynamic* and the variables holding these would have to be declared `dyn_var<T>`. Finally, this compiler will target NVIDIA GPUs. So the language for the second stage would be C++ for the host code and CUDA for the device code.

Since each of these components is implemented using multi-staging in BuildIt, the implementation of the EasyGraphit compiler does not require writing any compiler boilerplate or explicit analysis and transformations. This results in an immediate improvement in the lines of code required to implement the whole system. Before I get into each component, Table 6.1 shows the lines of code for each component in the original GraphIt DSL compiler and the EasyGraphit compiler, showing an about 11 times reduction in code size, which directly corresponds to the amount of effort the developer has to put in.

Frontend and Abstraction

Our implementation of the GraphIt compiler is implemented as an embedded DSL in BuildIt, which is a lightweight library in C++. Thus, all the types and operations are C++ classes and function/operator calls over those types, respectively. Thus, there is a necessity to implement a

traditional frontend with a lexer, parser, or even IR definitions. Figure 6.4 shows the definition of all the types and operators defined as the programming interface for our language. Some details of the implementation are omitted for brevity but the main user-facing API types in the language are - `struct Vertex`, `struct VertexData`, `struct GraphT`, `struct VertexSubset`, `struct PriorityQueue` for representing an individual vertex, an array of data associated with each vertex, an entire graph, a set of vertices to process and a priority queue object that holds a set of vertices in a priority order of implemented ordered applications like Single Source Shortest Path using the Delta Stepping algorithm. As shown in the Figure 6.4, these types are either `dyn_var<T>` types themselves or contain a combination of `dyn_var<T>` and `static_var<T>` types to perform data-flow analysis described ahead. The set of operators exposed to the user are - `vertexset_apply`, `struct edgeset_apply` with members `from`, `to`, `apply`, `apply_priority` and `apply_modified`. Finally, the language API also contains the scheduling types and functions, but those will be explained in the next subsection. Notice that the frontend component just counts the declaration of these types and functions since that is all that is exposed to the user. The actual operator definitions are part of the analysis and transformation section. The frontend section thus comes to a mere 55 lines of C++ code.

The original GraphIt compiler, on the other hand, implements a full-fledged frontend with a lexer, parser, and IR definitions. The IR definition has classes for 110 frontend AST nodes. Notice that the frontend IR is separate from the midend IR and has its own classes. The FIR has related classes, including visitors, lowering passes, and the actual parser. The various files in the `include/graphit/frontend` and the `src/graphit/frontend` and their lines of code are shown in Table 6.2. Just the frontend in the original GraphIt compiler relevant to the GPU component comes to a total of 8944 lines of C++ code, which is a whopping 162x larger than BuildIt embedded DSL compiler. This highlights the upfront cost of building DSL we saw in Chapter 1.

Scheduling Interface and Implementation

The scheduling interface in the BuildIt implementation of the GraphIt compiler defines scheduling classes as just a set of structs that can be passed as parameters to the operators. These scheduling objects have members that control various aspects of the implementation, like the load-balance strategy, the vertex set representation, or the direction or iteration, among others. The scheduling language interface in the original GraphIt compiler for the GPU backend is also similarly implemented as a C++ object that is actually inserted into the source code of the compiler, which, even though it makes it complex to compile, is easy to implement. Figure 6.5 shows the scheduling classes for the DSL implementation using BuildIt. The figure here just shows the main parts of the classes, while the actual implementation is a total of 151 lines of C++ code. The original GraphIt compiler has a similar API but also implements AST transformations to apply and transform the AST nodes according to these schedules. There are passes that progressively lower these nodes and apply the scheduling decisions to the MIR nodes in the compiler. Table 6.3 shows the various files that implement scheduling in the original compiler and their lines of code. The total comes to about 2240 lines of C++ code, which is once again 14.8x overhead in terms of lines of code.

Analysis and Transformation

The analysis and transformations are the bulk of the implementation of both implementations. Despite the lack of a traditional analysis and transformation pass infrastructure, this implementation

```
namespace graphit {
// Type declarations
struct Vertex {
    dyn_var<int> vid;
    // Analysis related fields
    enum class access_type {
        INDEPENDENT = 0,
        SHARED = 1,
        CONSTANT = 2
    };
    access_type current_access;
};
template <typename T>
struct VertexData {
        // Primary member to hold the vertex data
  dyn_var<T*> data;
        // Analysis related fields
  bool is_tracked;
  bool allow_dupes;
  VertexSubset *output_queue;
  graphit::SimpleGPUSchedule::frontier_creation_type frontier_creation;
};
struct PriorityQueue {
  PrioQueue pq;
  VertexData<int> *p;
  SimpleGPUSchedule* current_schedule;
};
// GraphT and VertexSubset are simply an extension to the dyn_var<T> type since
//No analysis is done for their members
struct GraphT: public dyn_var<builder::name<graph_t_name>> {};
struct VertexSubset: public dyn_var<builder::name<vertexsubset_t_name>> {};

// Operator declarations
typedef void (*vertexset_apply_udf_t) (Vertex);
void vertexset_apply(VertexSubset &set, vertexset_apply_udf_t);
void vertexset_apply(GraphT &edges, vertexset_apply_udf_t);
typedef std::function<void(Vertex, Vertex)> edgeset_apply_udf_t;
typedef std::function<void(Vertex, Vertex, dyn_var<int>)> edgeset_apply_udf_w_t;
struct edgeset_apply {
    void apply(GraphT &graph, edgeset_apply_udf_w_t udf);
    void apply(GraphT &graph, edgeset_apply_udf_t udf);
    void apply_priority(GraphT &graph, edgeset_apply_udf_w_t udf, PriorityQueue &pq);
    template <typename T>
    void apply_modified(GraphT &graph, VertexSubset &to, VertexData<T> &tracking_var,
    edgeset_apply_udf_t udf, bool allow_dupes = true);
    template <typename T>
    void apply_modified(GraphT &graph, VertexSubset &to, VertexData<T> &tracking_var,
    edgeset_apply_udf_w_t udf, bool allow_dupes = true)
};
}
```

Figure 6.4: Declarations for the types and operators declared as part of the BuildIt implementation of the GraphIt DSL.

Filename	Lines of Code
include/clone_apply_node_visitor.h	36
include/clone_for_stmt_node_visitor.h	36
include/clone_loop_body_visitor.h	35
include/error.h	249
include/fir.h	1888
include/fir_context.h	35
include/fir_visitor.h	349
include/frontend.h	30
include/parser.h	263
include/scanner.h	36
include/token.h	166
src/clone_apply_node_vistor.cpp	35
src/clone_for_stmt_node_vistor.cpp	36
src/clone_loop_body_visitor.cpp	33
src/error.cpp	104
src/fir.cpp	1,313
src/fir_visitor.cpp	534
src/frontend.cpp	43
src/parser.cpp	2,851
src/scanner.cpp	573
src/token.cpp	299
Total	8,944

Table 6.2: The different files under the original GraphIt compiler frontend directory and their number of lines

Filename	Lines of Code
src/high_level_schedule.cpp	975
src/low_level_schedule.cpp	396
include/high_level_schedule.h	275
include/low_level_schedule.h	265
include/gpu_schedule.h	329
Total	2,240

Table 6.3: The different files under the original GraphIt compiler schedule directory and their number of lines

```
// Simple GPU Schedule class
struct SimpleGPUSchedule: public Schedule{
    static int default_max_cta;
    static int default_cta_size;
    // Enums defining various options
    enum class direction_type { PUSH, PULL};
    enum class pull_frontier_rep_type { BITMAP, BOOLMAP };
    enum class frontier_creation_type { SPARSE, BITMAP, BOOLMAP };
    enum class deduplication_type { DISABLED, ENABLED };
    enum class deduplication_strategy_type { FUSED, UNFUSED };
    enum class load_balancing_type { VERTEX_BASED, TWC, TWCE, WM, CM, STRICT, EDGE_ONLY };
    enum class edge_blocking_type { BLOCKED, UNBLOCKED };
    enum class kernel_fusion_type { DISABLED, ENABLED };

    // Actual members holding the configuration options
    direction_type direction;
    pull_frontier_rep_type pull_frontier_rep;
    frontier_creation_type frontier_creation;
    deduplication_type deduplication;
    deduplication_strategy_type deduplication_stategy;
    load_balancing_type load_balancing;
    edge_blocking_type edge_blocking;
    int block_size;
    kernel_fusion_type kernel_fusion;

    int max_cta;
    int cta_size;
};
// Hybrid GPU Schedule class
struct HybridGPUSchedule: public Schedule{
    public:
    Schedule* s1;
    Schedule* s2;
    // The threshold is a dynamic parameter to allow for runtime scheduling
    dyn_var<float> *threshold = nullptr;
    float static_threshold;
    HybridGPUSchedule(Schedule &_s1, Schedule &_s2) {
        s1 = &_s1; s2 = &_s2;
    }
    void configThreshold(float t) {
        threshold = nullptr;
        static_threshold = t;
    }
    void bindThreshold(dyn_var<float>&);
};
```

Figure 6.5: The **`SimpleGPUSchedule`** and the **`HybridGPUSchedule`** classes in the BuildIt implementation of GraphIt DSL

of the DSL on top of BuildIt implements all the operators through progressive lowering and specialization. The schedules and the operators are expanded into actual low-level C++ code for Host and GPUs, as shown in the example of the `apply` operator. Besides that, the compiler also uses the methodology explained in Chapter 5 to perform a data-flow analysis to automatically insert atomics. This data-flow analysis keeps track of each Vertex variable if it is `SHARED` between multiple threads, `INDEPENDENT`-ly assigned to a single thread, or is a `CONSTANT`. This tracker property is then used to determine whether an array access based on this variable can result in a race condition and inserts atomics accordingly. This data-flow analysis is implemented as part of the `Vertex` type and is used when indexing inside the `VertexData` type. Besides these analyses, the compiler also implements GPU Kernel Fusion using the technique described in Chapter 5. Being the bulk of the DSL implementation, the total lines of code required to implement this component are about 1,320 lines of C++ code, which mainly includes the implementation of each operator like `apply`, `apply_modified`, `apply_priority`, which each implement several options based on scheduling and runtime decisions. The lines of code for both the EasyGraphit implementation and the original compiler are shown in Table 6.4 and Table 6.5, respectively. The original GraphIt compiler also includes the implementation of the midend IR, where the actual transformations are performed. Each transformation in the original GraphIt compiler is so large because it contains a lot of boilerplate due to the pass infrastructure. The total lines of code in the original GraphIt compiler are about 10,358 lines of C++ code as opposed to the 1,318 lines of C++ code in the EasyGraphit compiler, which is about 7.85 times the overhead.

include/operators.h	183
include/graphit_types.h	449
src/operators.cpp	674
src/graphit_types.cpp	12
Total	1,318

Table 6.4: The different files under the EasyGraphit compiler implementation for implementing operator lowering and their number of lines

Code Generation for Host and Device

Finally, the last component of the GPU compiler for GraphIt embedded in BuildIt is the code generator. Technically, the compiler doesn't have any implementation for the backend since the code generation is completely handled by the C, C++, and CUDA code generator in BuildIt. However, the GPU extensions to BuildIt using the `"CUDA_KERNEL"` annotation were first added to BuildIt with this implementation, which was about 550 lines of C++ code change to BuildIt itself. The original GraphIt compiler, on the other hand, has a complete Host and GPU side code generator which prints out strings by lowering the midend IR after analysis. The total implementation of this backend is about 2188 lines of C++ code, which is 3.97 times the overhead.

Finally, all the numbers combined, the EasyGraphit compiler comes to a total of 2021 lines of code, while the original GraphIt compiler implemented using traditional compiler techniques is about 22,092 lines of code, which is a total saving of 11 times, over a magnitude in lines of code

Filename	Lines of Code
apply_expr_lower.h	56
atomics_op_lower.h	74
change_tracking_lower.h	76
field_vector_property.h	26
frontier_reuse_analysis.h	38
gpu_change_tracking_lower.h	41
gpu_priority_features_lowering.h	57
gpu_vector_field_analyzer.h	50
intersection_expr_lower.h	49
label_scope.h	63
merge_reduce_lower.h	56
midend.h	37
mir.h	1,742
mir_context.h	475
mir_emitter.h	190
mir_lower.h	24
mir_metadata.h	53
mir_printer.h	46
mir_rewriter.h	193
mir_visitor.h	297
par_for_lower.h	50
physical_data_layout_lower.h	67
priority_features_lowering.h	174
priority_queue_frontier_reuse.h	10
udf_dup.h	26
var.h	46
vector_field_properties_analyzer.h	108
vector_op_lower.h	35
vertex_edge_set_lower.h	26
while_loop_fusion.h	24
apply_expr_lower.cpp	414
atomics_op_lower.cpp	332
change_tracking_lower.cpp	202
frontier_reuse_analysis.cpp	87
gpu_change_tracking_lower.cpp	179
gpu_priority_features_lowering.cpp	120
gpu_vector_field_analyzer.cpp	164
intersection_expr_lower.cpp	49
merge_reduce_lower.cpp	77
midend.cpp	22
mir.cpp	1,092
mir_context.cpp	7
mir_emitter.cpp	1,102
mir_lower.cpp	108
mir_printer.cpp	67
mir_rewriter.cpp	481
mir_visitor.cpp	440
par_for_lower.cpp	34
physical_data_layout_lower.cpp	179
priority_features_lowering.cpp	392
udf_dup.cpp	39
vector_field_properties_analyzer.cpp	237
vector_op_lower.cpp	231
vertex_edge_set_lower.cpp	59
while_loop_fusion.cpp	35
Total	10,358

Table 6.5: The different files under the GraphIt compiler for implementing the midend IR, analyses, and transformations.

saved. Furthermore, as described, the implementation in the EasyGraphit compiler is similar to what would be implemented inside a high-performance graph library, thus requiring even less effort to make the transformation.

6.1.3 Performance Evaluation

Finally, we compare the performance of the code generated from the compiler written using BuildIt and compare it with the performance of the code generated by the original state-of-the-art GPU GraphIt compiler [25].

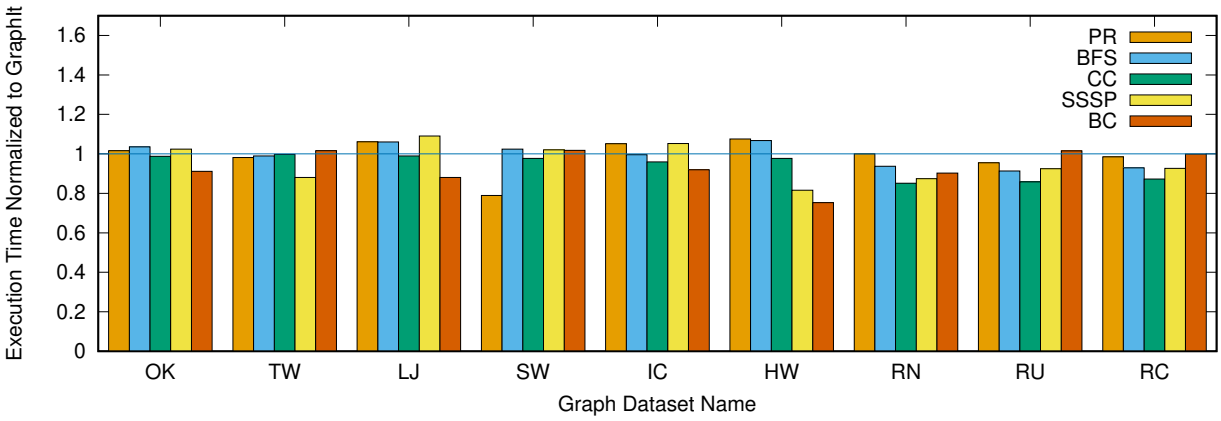


Figure 6.6: Relative execution times of the code generated from the EasyGraphit compiler for each of the 5 applications - BFS, PR, SSSP, CC, and BC on the 9 datasets normalized to the execution time of the code generated by the original GraphIt compiler.

We benchmark 5 applications - PageRank (PR), Breadth First Search (BFS), Single Source Shortest Path with Delta Stepping (SSSP), Betweenness Centrality (BC), and Connected Components (CC). These applications include a mix of topology-driven, data-driven, and priority-driven algorithms that fully stress the various aspects of the compiler. Similarly, we use 9 different graph datasets with these applications. These datasets include a mix of power-law degree and bounded degree high-diameter graphs, which have different characteristics and sparsity patterns. We run our evaluations on an NVIDIA Volta V100 (32 GB memory, 4MB L2 cache, and 80 SMs) GPU. Both for the original GraphIt and the EasyGraphit compiler, we tune the best schedule for each algorithm and graph input for a fair comparison (since the programming interface and the scheduling language for both are similar, these schedules are very close to each other).

Figure 6.6 shows the execution times of the code generated from the EasyGraphit compiler for each of the 5 applications on each of the 9 datasets, normalized to the execution time of the code generated by the original compiler. In this figure, a value of 1 would mean that both implementations produce code that runs in an equal amount of time. A bar shorter than shows the generated code by the EasyGraphit compiler is faster, while a bar longer implies the original compiler's generated code is faster. As we can see in most cases, the performance of the code generated by the EasyGraphit compiler is faster or has almost the same performance. The performance is slightly slower on some of the graphs, but the overall geometric speedup is 1.03x with the maximum slowdown of 8.3%. The paper [20] discusses these results and the explanation for each in detail.

The overall conclusion is that BuildIt can help reduce the effort for implementation of a compiler by more than 10x while still achieving the same performance. I have thus, through this case study, demonstrated that BuildIt is as close to the ideal choice of abstraction for generating high-performance code without having to deal with any compiler-ish code while also significantly reducing the amount of effort required.

6.2 NetBlocks: Staging Layouts for High-Performance Custom Host Network Stacks

In this section, I will describe the design and implementation of NetBlocks [24], a compiler for generating ad-hoc networking protocols tailored to application needs and deployment scenarios. Networking is a domain that serves as a backbone for many other high-performance domains like distributed systems, databases, and even large-scale machine learning training. As a result, improving the performance of how fast data can be moved between hosts is important for the performance of the overall system. However, these network stacks have to meet several constraints, like making sure packets are not lost or corrupted or arrive in the same order they were sent, especially when being sent over multiple routes, which may or may not be applicable to all applications. The confluence of performance requirements with domain-specific constraints presents a unique opportunity for applying compiler techniques to generate network protocols. The modular nature of the protocols and the low-level nature of existing implementations in languages like C and C++ make BuildIt an ideal choice for implementing this compiler. Let us look at the problem in more detail.
For a very long time, network applications and deployments have relied heavily on the classic UDP, TCP, and IP protocols. The design of these protocols was made keeping in mind the requirements of applications that were around during the birth of the internet. Packets needed to be routed long distance over multiple hops, physical networks were lossy and susceptible to packet corruption and congestion, and generally not predictable. The implementations of these protocols have been hand-optimized over decades to provide the best possible performance. A lot of research has also been invested in improving certain parts of the algorithms, like better congestion control algorithms, better routing algorithms, etc., but the overall structure of these all-or-nothing algorithms remains the same.

However, recent times have seen a massive paradigm shift that has forced network researchers to rethink protocol design. Network applications have permeated new domains like (*i*) IoT that are constrained by extremely low-power, (*ii*) underwater robotics that are constrained by extremely low bandwidth [94, 124] (*iii*) isolated dedicated networks for Machine Learning training, where the overall performance also depends on the best utilization of the network bandwidth, and (*iv*) AR/VR applications that are driven by extremely tight latency guarantees.

These new domains and their environments offer vastly different network properties and constraints that cannot be met by legacy TCP and UDP protocols. For example, an underwater sensor network of robots communicating through acoustic signals with very limited bandwidth cannot afford the 42-byte overhead of IP/UDP network headers. At the same time, these applications don't require all features from the protocols like reliability, in-order delivery, checksumming, and congestion control, but only a subset depending on the application and physical network properties. Recent years have also seen massive innovations in network hardware, which offer microsecond-scale latency and 100s of Gigabits of throughput. Naturally, the bottlenecks are shifting from the hardware limitations to the network protocol design and implementation [28]. The end-to-end latency depends not only on the number of bits transmitted but also on the number of cycles spent in host-side processing in implementing the protocol logic on the hosts and other network devices.

The logical solution is to create and deploy custom ad-hoc protocols tailored to the needs of both the applications and the environment. Unlike the all-or-nothing approach of TCP and UDP when it comes to features available, network application developers should be able to pick and choose

features and their customization to get the best network performance while meeting the critical requirements of the applications. An example of such a custom protocol is Google's QUIC [113], which combines basic reliability on top of UDP with Transport Layer Security (TLS) to suit web applications. However, as seen with all other DSLs, writing and optimizing custom protocol implementations, especially with a combination of features, is daunting, which has hindered the widespread deployment of such ad-hoc protocols. The QUIC protocol, for example, requires effort from hundreds of developers at Google for its development and maintenance. Furthermore, changes in deployment scenarios also require continuous development effort. For example, a custom protocol used in a network with 16 nodes uses 4 bits to identify source and destination hosts. However, when this deployment is scaled to a network with 32 nodes, 5 bits would be required to represent the hosts, which would completely require rearranging the headers to optimally pack the bytes while meeting the alignment and size requirements of the other fields.

Naturally, a compiler solution that can minimize both the upfront cost of creating simple protocols and also allow for adding new features with ease is necessary. I will explain two scenarios from two completely different areas to motivate this problem better.

6.2.1 Motivating Examples

In this section, I will present two scenarios that motivate the need for custom-tailored protocols and the performance trade-offs they offer.

Video Conferencing Application

The user experience of video and audio conferencing applications that have become ever more necessary today depends heavily on the latency of the communication, since even a small delay can cause a lag in the voice or the video. The popular conferencing applications and the stacks they use are heavily optimized to minimize latency.

These applications, however, also have unique characteristics that legacy UDP/TCP protocols are not designed to handle. For example, video conferencing does not care about reliability and can tolerate a few packets being dropped. It can also tolerate a few bits in the packet payload being corrupted, since these corruptions might lead to only minor distortions in the audio/video. Since the network exchange happens at a fixed rate, the application may not even care about congestion control. These requirements might convince the developer to lean towards a protocol like UDP, which is very lightweight. However, video conferencing applications rely critically on some notion of in-order delivery. If some packets arrive out of order, it could lead to voice and video getting completely jumbled. The UDP protocol, unfortunately, has no notion of in-order delivery or even a connection. Even if the developer decides to use TCP, the notion of in-order delivery in TCP often holds packets back till they arrive in order, further compromising latency.

Consequently, the developers must implement a custom in-order delivery mechanism on top of UDP, which increases implementation complexity. Furthermore, if we peer at the problem carefully, we realize that even though this application is tolerant to corruption in the payload, it might not be tolerant to corruption of certain control headers in the exchange. To keep the overhead of computing and verifying the checksums, the developer would have to specify only a certain part of the headers and payload to be checksummed. Handling this manually at the application layer would further increase developer effort and implementation complexity. The key requirement here is that the

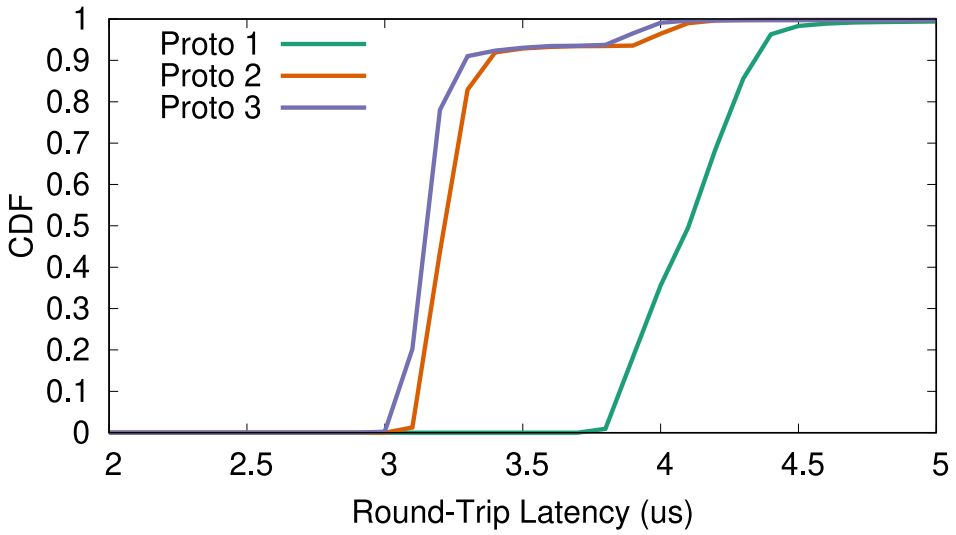


Figure 6.7: CDF of round-trip latency of 10000 packets for three custom protocols with decreasing number of features. Proto 1 has all features, including reliability and in-order delivery. Proto 2 disables reliability and uses a simpler version of in-order delivery. Proto 3 is bare bones and only restricts checksumming to headers.

developer should be able to quickly Experiment with different features and their performance to decide what fits their specific application needs.

Figure 6.7 shows the CDF plot of round-trip latency for 10,000 packets of three custom protocols. Protocol 1 has full reliability and in-order delivery. Protocol 2 disables reliability, uses a simpler version of in-order delivery, and restricts the checksumming only to the network headers. Protocol 3 removes all notions of reliability, in-order delivery, or checksumming. We can see that the Protocol 2 which is almost as performant as the bare-bones Protocol 3, has about 25% median lower latency than Protocol 1, which adds unnecessary features. With a carefully selected protocol, the developer only pays for what they need in terms of performance.

Underwater Robotics Sensing

As another motivating example, we look at a remote-sensing robot deployed underwater that gathers and sends sensor data to the base station. The key constraint of this environment is that the communication is done through audio waves traveling through water, and not typical EM waves in the air. This unique environment characteristic means that the robot and the base station operate at very low bandwidth, typically tens to hundreds of bits per second. The network stack needs to minimize the number of bits transferred in every way possible, not only to save network bandwidth but also to minimize the device's power utilization.

For an application where the robot gathers and sends sensor data, the payload size is typically as small as 16 bits. As the actual payload size is reduced, the overhead from packet headers starts to dominate. Even the simplest UDP protocol running on top of IP running on top of an Ethernet-like protocol requires a 42-byte header with many useless or redundant fields. Figure 6.8 a) shows the headers for the three protocols (Ethernet, IP, and UDP) stacked on top of each other. The fields

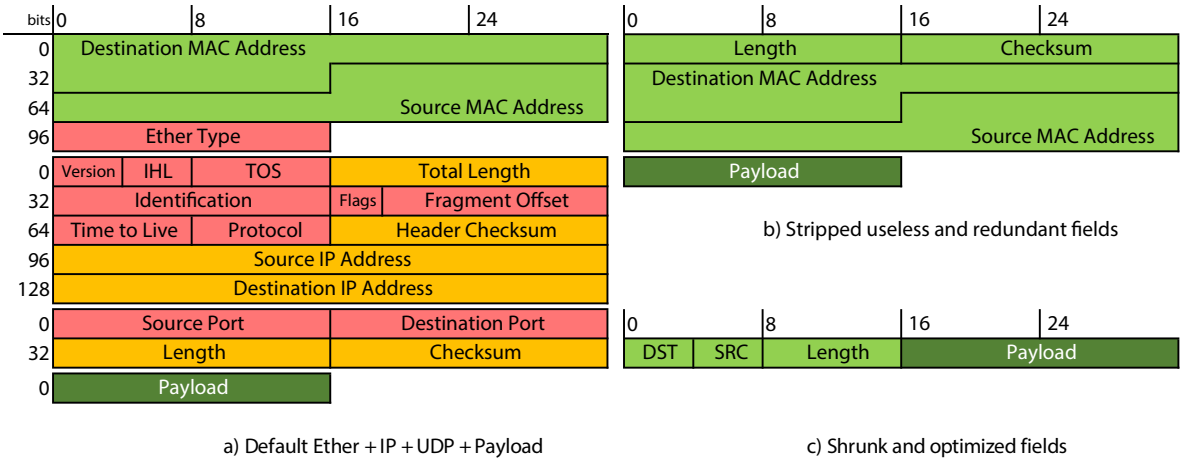


Figure 6.8: a) Default Ethernet + IP + UDP headers with 50 bytes. b) Protocol with useless and redundant fields stripped down c) Custom protocol with fields shrunk to fit the required deployment. 2 bytes of payload

in red correspond to bits that have no utility in this scenario. The fields in orange, like length and checksum, are redundant since multiple layers implement them. This is an example of overhead due to the independent development of protocol layers. A simple customization could strip these fields from the headers. Figure 6.8 b) shows such a minified protocol that requires only *16* bytes.

We can compress this further. In this header, 6 bytes are used to identify the source and destination MAC addresses of the robots. However, if our deployment has only 16 robots, 4 bits would be enough to identify all the hosts. Similarly, if there is a cap on the size of the payload, the number of bits for the size can also be shrunk, giving us a protocol shown in Figure 6.8 c) that only uses *2* bytes for the headers. Suppose our deployment changes and our fleet has 32 robots instead of 16. Consequently, the headers and protocols need to be adjusted to use 5 bits instead of 4. These could also be bumped to 8 bits instead of 5 for alignment reasons. Without a compiler-based solution, the developer would have to change these protocols with every new deployment manually.

The two motivating examples have shown that modern applications require custom host network stacks to meet their tight latency and bandwidth requirements. Moreover, the changing requirements of the features and implementations based on application needs and deployment scenarios make a hand-curated and optimized solution infeasible. Just like every performance-critical domain, the runtime cost of implementing simple protocols and composing them at runtime is also too high. Naturally, a compiler solution is the most ideal to address the changing requirements while guaranteeing performance. BuildIt is also a great fit for this domain since it offers composing independently written features without little to no runtime cost and also allows for creating custom layouts as needed by these protocols using the Layout Customization Layer described in Chapter 5. Finally, and probably most importantly, a compiler written with BuildIt allows for easily extending the system to add more features like encryption, compression, or even application-specific features like TLS handshake and so on at the protocol layer itself with ease. Since BuildIt uses a multi-stage programming interface, these new features can be added by simply writing a library-like implementation as opposed to a compiler implementation, which is much more suitable for network engineers and researchers. I call this compiler for generating network protocols, NetBlocks. I will

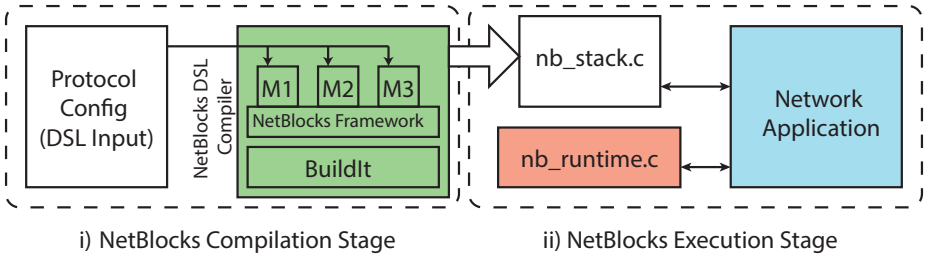


Figure 6.9: The overall architecture of the NetBlocks DSL compiler with the compilation phase, where the NetBlocks DSL input is used to generate a specialized stack that is linked with the application in the execution phase.

explain the design of the NetBlocks input language and the overview of the compiler next.

6.2.2 NetBlocks Language and Compiler Overview

In this section, I will first describe the overall architecture of the NetBlocks compiler, followed by the programming interface it offers.

Overall Workflow

Before looking at the internals of the compiler, I will describe the overall workflow of using the NetBlocks compiler. The basic idea of the NetBlocks compiler is to break down network protocols into individual features like reliability, in-order delivery, checksumming, routing, and so on, which the application developer can pick and choose. The execution of the compiler and the execution are separated into two distinct phases, as shown in Figure 6.9. During the compilation phase, the NetBlocks compiler accepts DSL input which specifies the choice of features, their configurations to choose what exact flavor of each is required, and constraints about deployment, including the number of network nodes, routing constraints, and even ranges of specific values like packet length, etc. These inputs go into the NetBlocks compiler that is implemented on top of BuildIt's `static_var<T>` and `dyn_var<T>` types. The output of the NetBlocks compiler is the source code in C for the custom protocol specified. This specialized protocol gets compiled and linked into the application along with the NetBlocks runtime library, which has basic utility support for managing data structures and, most importantly, for communicating with the network hardware. The compiled and linked application then runs during the NetBlocks execution phase, where it actually sends and receives packets. Furthermore, network developers can also add their own features to augment the compiler during the NetBlocks compilation phase to generate a stack with even more features.

NetBlocks Compiler Architecture

The NetBlocks compiler and runtime are divided primarily into three parts - The *Framework*, the *Modules*, and the *Runtime Library*

The *Framework*: The *Framework* provides the core functionality of managing the state of the compiler, also generates code to manage state at the runtime - including managing allocation connection objects for new connections, managing coming in and out of the system. Providing

the interface to the application. At compile time, the *Framework* performs the primary function of hosting all the *Modules* configured by the users and calling them in order to generate code for various parts of the protocol. The *Framework* also implements the Layout Customization Layer as described in Chapter 5 for allowing the *Modules* to create, modify, and access fields in the packets. Finally, the *Framework* also calls the BuildIt API functions like `extract_function_ast` and the C Code Generator to generate the code for the network stack.

The *Modules*: The *Modules* are building blocks of the network protocols that implement an individual function like reliability, in-order delivery, checksumming, routing, identifying which connection packet belongs to, and so on. Modules perform two critical functions: *i*) they describe the logic required to implement the features to the NetBlocks compilers so that the logic can be combined with the other features to generate the whole protocol, and *ii*) They create, modify, and access the field required for the implementation of the protocol in both the packet and the auxiliary data structures. For example, the reliability module is responsible for creating and managing the sequence number and acknowledgment sequence number in the packets and data structures like the last received sequence number and the redelivery buffer in the connection object data structure. Finally, it is also the reliability module's responsibility to implement the logic to add a sequence number to every outgoing packet, send ACKs when a packet comes in, and retransmit packets in case of a timeout of a duplicate ACK. The modules can also accept customization parameters from the user during the NetBlocks compiler stage to choose different strategies for the implementation. For example, the inorder module has three strategies: *i*) No Inorder: where packets are delivered in the order they arrive, *ii*) Drop out of order: Where packets that arrive out of order are dropped, and packets are delivered only in the increasing order of sequence numbers. *iii*) and Hold Forever: where packets that arrive out of order are held in an in-order buffer till the missing packets arrive. This is similar to the typical in-order delivery in the TCP protocol. Table 6.6 shows the list of the modules currently implemented as part of the NetBlocks framework and the configuration options they offer. Just by picking and choosing features and their configuration options, a developer can generate 864 different unique protocols.

The *Runtime Library*: The *Runtime Library* is all the non-generated code that forms part of the network stack that runs during the execution phase. The runtime library provides functions to create and manage data structures at runtime. This includes structures like packet queues, where packets are held in the connection object before being delivered to the user application, or the redelivery queue, where the packets are held to be re-sent in case of packet loss. The *Runtime Library* also provides functions for creating and managing timers for various events. Finally, the *Runtime Library* provides an API and implementation for interacting with the actual networking hardware, like the Network Interface Card (NIC). As opposed to being generated like the rest of the implementation, the *Runtime Library* is a set of handwritten functions that need little modification. Since this code is not generated, it doesn't need to use the BuildIt types and new functionality like support for interacting with a new type of hardware can be added easily by adding a new function. In the next section, I will explain the NetBlocks compiler and how its different components are implemented.

6.2.3 Implementing the NetBlocks Compiler

As explained in the previous section, the NetBlocks compiler has three main components - The *Framework*, the *Modules*, and the *Runtime Library*. Since the goal of the compiler is to generate a low-level implementation based on the choices provided by the user regarding the features selected

Modules	Description	Configurations	Count
Identifier Module	Identifies which flow or connection a packet belongs to	Local identifiers (MAC)/ Global identifiers (IP), App identifiers	4
Routing Module	Implements Global identifiers like IP address for Routing	Enabled/Disabled + ranges	2
Checksumming Module	Implements checksumming for headers and payloads	No/Header only/whole packet checksumming	3
Reliability Module	Implements reliable delivery	No Reliable/Reliable	2
Inorder Module	Implements In-order delivery	No Inorder/Drop out of order/Hold forever	3
Payload Module	Manages setting and retrieving payload	-	1
Signaling Module	Implements TCP like handshake for connection establishment	Signal connection/Don't signal connection	2
Compression Module	Compresses packets/payload for bandwidth optimizations	No/Full packet/Payload compression	3
Total			864

Table 6.6: Currently implemented list of *Modules* in the NetBlocks compiler, their description, and the features they offer.

and their configuration options, the deployment constraints, and their effects on the low-level packet layout. As part of the generated code, the compiler needs to generate code for the logic that runs when a new connection is established, when a new packet is to be sent out, when a packet comes in, and when a connection is destroyed. Apart from this, the compiler also needs to create the code to initialize all data structures for a protocol when it is initialized. We notice that the choice whether a particular feature is enabled and its configuration potentially changes the behavior of each of these separate events. Similarly, with the changing features and deployment options, each of the fields in the packet would also be accessed differently. For example, if a field, like say the length field, is stored as a 32-bit integer in the packet, it would be read/written by a single load/store. However, suppose the range of the length field has been changed to 64 to 128 bytes only; it can be represented in as few as 6 bits. This means reading this field would require a mask operation, potentially followed by a shift operation, and finally followed by an addition with 64. The store operation would similarly be the inverse of this. Notice that everywhere in the generated implementation, the field is accessed, the generated code would have to change based on the inputs. Finally, we want the developers to be able to add new features as modules quickly without having to write any compiler code so that they can combine these new features with these existing features at the stack level itself. Thus, we come up with the following three requirements for the implementation of the NetBlocks compiler:

1. Global changes to the generated protocol logic based on the selection of a feature while maintaining high-performance

2. Global changes to the layout and the code to access it based on the selection of features and

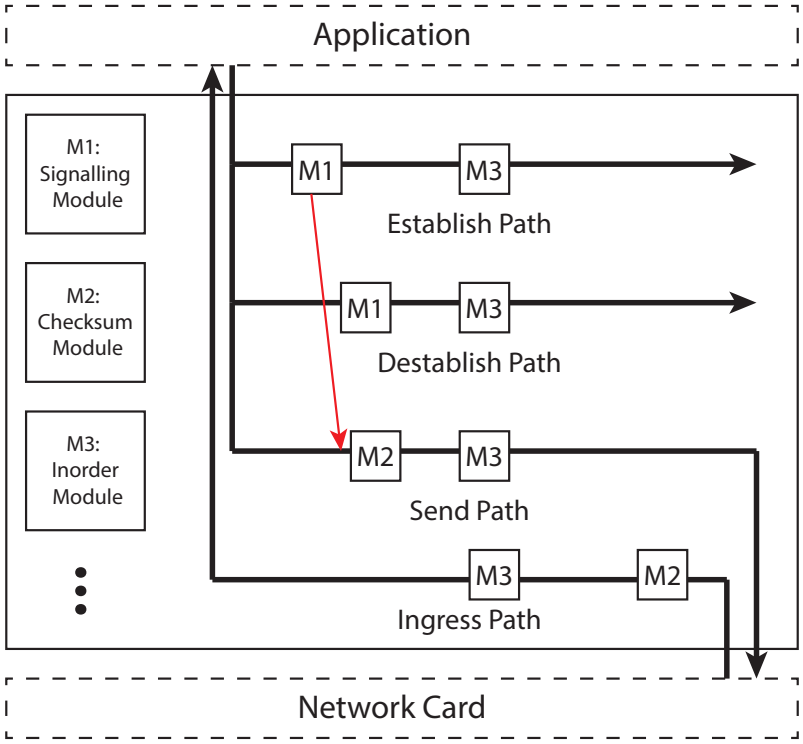


Figure 6.10: A block diagram showing the *Framework* defining various *Paths* and the various registered *Modules* inserting their hooks into them. The hooks change the behavior of what happens when a path is run. Paths can also invoke other paths.

deployment constraints, without having to rewrite any code in the module logic

3. Ability to extend the protocols that the NetBlocks compiler can generate by adding new modules in a network domain, expert-friendly manner, without having to write any compiler code.

I will explain how a High-Performance Aspect-Oriented Design on top of BuildIt, with the use of BuildIt's Layout Customization Layer, helps NetBlocks achieve the above goals.

High-Performance Aspect Oriented Programming System with BuildIt

We observed above how a change in the choice of one feature leads to global changes across all parts of the generated code. This kind of parameterization can be achieved by writing the whole stack as one monolithic block and spreading out conditions that are dependent on parameter choices everywhere. While this approach works, it is extremely hard to manage since the same configuration parameter or a combination of parameters would need to be checked everywhere. In the past, such a problem has been solved using a programming pattern known as Aspect-Oriented Programming (AOP). Aspect-oriented programming allows breaking down the logic into so-called distinct *concerns*. These concerns are allowed to augment the behavior of existing code without modifying the code itself. The simplest and most modular way of implementing aspect-oriented programming in a language like C++ is done through the use of virtual hooks that can be inserted

into various paths.

We observe that AOP is a great fit for the modular design of features in NetBlocks. Individual network features like reliability and in-order delivery can be broken down into individual concerns, which can insert hooks into various events that can be invoked at runtime. These hooks can then change the behavior of the code path taken during these events. Figure 6.10 shows the four distinct paths that are part of the NetBlocks framework, including the Establish Path, Deestablish Path, Send Path, and the Ingress Path. The registered modules on the left insert hooks into these paths to add logic during execution. These modules and hooks can terminate the path at any point to drop the packet or continue for the next hook to be invoked. These hooks can also invoke other paths as part of their execution. For example, when the establish hook for the signaling module is invoked on the establish path, it can invoke the send path to send a packet to the other side to signal the start of a new connection. In the same way, the reliability module can invoke the send path from its ingress hook to send an ack. Figure 6.11a shows the implementation of the `module` base class declaring virtual functions for the various hooks that the various modules can implement. The default implementations of all these hooks simply return `HOOK_CONTINUE` to signal that the next module must be called. The figure also shows the implementation of the `reliability_module` that extends the `module` base class and overrides the `hook_send` and `hook_ingress`. The `hook_send` inserts the sequence numbers in all packets being sent out and increments the last sequence number stored in the connection object. Similarly, the module also overrides the `hook_ingress` to call `send_ack` to send acknowledgments for each packet it receives. The actual implementation of the reliability module has more conditions and more logic. This is just an example for brevity. Finally, the `run_send_path` function that invokes all the hooks is shown. It iterates through all the hooks and invokes them one by one. If any hook returns `HOOK_BREAK`, the execution of the path is terminated. Such `run_*_path` functions are implemented for other paths in a similar way. Such a design allows separating concerns in a modular way where all the logic and implementation related to a feature is wrapped in a single class, and entire objects of classes can be removed or inserted to enable/disable the feature. However, this abstraction and modularity come at a cost. If the actual network stack is implemented this way, the send path would be invoked for each packet being sent, and there would be multiple virtual dispatches followed by several conditions for every packet, severely affecting the latency and throughput.

However, we make an observation that the knowledge of which modules are installed and how the virtual calls should resolve is known entirely during the compilation phase. Thus, if we use BuildIt's multi-staging, we can partially evaluate this code to get an optimized implementation specialized based on the specific modules enabled and the way they are configured. Figure 6.11b shows the staged version of the modules, where the functions look exactly the same except for the argument types, like the actual packet and the connection object are made *dynamic* since they depend on runtime parameters. Also notice in the `run_send_path`, the iteration over the registered modules is done using *static* variables, and the status values returned from the hook functions are also *status*. Thus, not only would the virtual functions be resolved during the *static* stage (corresponding to the compiler phase of NetBlocks), but the status values and the condition to terminate the path would also be evaluated in the *static* stage, eliminating the overhead of modularity and abstraction. Furthermore, the modules themselves can also have conditions based on configuration parameters, allowing for configuring and optimizing the features themselves based on feature parameters and deployment constraints. Thus, a classic technique like aspect-oriented programming combined with multi-staging allows us to build true modular abstractions with zero cost.

This design not only addresses the requirement of being able to make global changes to the protocol logic by changing some parameters locally while maintaining high performance, but also allows for easy extensibility. Adding a new feature to the NetBlocks compiler is now as easy as implementing a new module class with overridden hook functions. Furthermore, since these modules are written using BuildIt, writing modules that generate code from the compiler does not require writing any compiler-ish code but simply how the feature would be implemented in a library, making extending the NetBlocks compiler to add new features feasible for network engineers. This also allows us to quickly port implementations from popular network libraries already written in C or C++ into NetBlocks without requiring many changes.

Applying Layout Customizations to Packets

In the NetBlocks compiler, as the chosen features and the deployment-related parameters are changed, headers in the packet can be removed, resized, or even moved around for the best performance. However, since the configuration parameters are only known during the execution of the compiler and not during the module implementation time, it is impossible to write a static implementation of structs, bit fields, and access functions. Furthermore, the access function of one field changes depending on the changes to the field before and around it. A similar problem applies to auxiliary data structures that manage runtime state, like the `conn_t` object that keeps track of each open connection. Optimizing layouts based on features and deployment scenarios requires programmatic access to data structures and data layouts. We discussed this problem and a possible solution for this in Chapter 5 using the Layout Customization Layer.
We apply the techniques discussed in Chapter 5 for two separate problems:

Optimizing the Connection Object: The connection object, defined as part of the NetBlocks code-base and wrapped in `dyn_var<T>` and thus part of the generated code, keeps track of the state of a specific open connection. This includes fields like the 4-tuple identifying the connection (source and destination IP addresses and the source and destination port numbers) and other fields like the last sent sequence number, the last received in-order sequence number, the queue of packets ready to be delivered to the application layer, and so on. Naturally, the fields present and their types depend on the features and their configuration. To allow such customization in the *static* stage of BuildIt, we use the `dynamic_object_base<T>` type introduced in Chapter 5. This object allows modules to insert fields with programmable types at module configuration and initialization time and access those fields by string names inside the hooks. The generated code thus would now have a programmatically generated `conn_t` object with fields of appropriate types, and the generated code would access those fields directly by member name without the overhead of string lookups at runtime.
Optimizing the Packet Layout: The second most important data structure in the generated network protocol is the packet layout itself. Just like the `conn_t` object, the exact fields present in the packet depend on the features chosen and their configuration. Furthermore, these fields and their sizes also depend on the deployment constraints, such as the range of individual fields like IP ranges and maximum sequence numbers, and so on. Since the optimized layout can have fields that might be represented by a mere handful of bits, the packet layout requires more fine-grained control as compared to the `conn_t` object for obtaining the best latency. Thus, we use the Layout Customization Layer introduced in Chapter 5 instead. This layer allows modules to insert and manage individual fields during initialization time. Then the framework can optimize the entire packet layout by

```
struct module {
  virtual
  status hook_establish(conn_t*) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_destablish(connt_t*) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_send(conn_t*,
        packet_t*) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_ingress(packet_t*) {
    return HOOK_CONTINUE;
  }
};
struct reliability: public module {
  status hook_send(conn_t* c,
        packet_t* p) {
    p->seq_num = c->new_seq_num++;
    return HOOK_CONTINUE;
  }
  status hook_ingres(packet_t* p) {
    send_ack(p->seq_num);
    return HOOK_CONTINUE;
  }
};
...
void run_send_path(conn_t* c,
        packet_t* p) {
  for (module* m: all_modules)
    if (m->hook_send(c, p) == HOOK_DROP)
      break;
}
```

(a)

```
struct module {
  virtual
  status hook_establish(dyn_var<conn_t*>) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_destablish(dyn_var<connt_t*>) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_send(dyn_var<conn_t*>,
        dyn_var<packet_t*>) {
    return HOOK_CONTINUE;
  }
  virtual
  status hook_ingress(dyn_var<packet_t*>) {
    return HOOK_CONTINUE;
  }
};
struct reliability: public module {
  status hook_send(dyn_var<conn_t*> c,
        dyn_var<packet_t*> p) {
    p->seq_num = c->new_seq_num++;
    return HOOK_CONTINUE;
  }
  status hook_ingress(dyn_var<packet_t*> p) {
    send_ack(p->seq_num);
    return HOOK_CONTINUE;
  }
};
...
void run_send_path(dyn_var<conn_t*> c,
    dyn_var<packet_t*> p) {
  for (static_var<module*> m: all_modules)
    if (m->hook_send(c, p) == HOOK_DROP)
      break;
}
```

(b)

Figure 6.11: Implementation of *Modules* as aspects inserted into the various code path as hooks a) without and b) with BuildIt's multi-staging.

optimizing for metrics like total bits required. The optimizations can be applied separately to fields grouped into layers, so that compatibility can be maintained with existing layers like Ethernet by manual control on a subset of headers in case compatibility is required with legacy switches and NICs. The modules can then access the fields in a packet through the optimized layout object, generating the most optimized code for accessing the fields. A small change in the parameter can generate an entirely different layout without having to manually modify structs or bit-fields.

Thus, the techniques introduced in Chapter 5 for optimizing data structures and data layouts in BuildIt help us satisfy the requirement of NetBlocks of being able to globally change layouts based on individual feature and deployment changes.
Thus, by combining high-performance aspect-oriented programming written with BuildIt's multi-staging and by optimizing packet layouts and other runtime data structures, NetBlocks is able to generate optimized code for various application needs and deployment scenarios. In the next section, I will describe the performance evaluation to highlight the benefits of careful feature selection in NetBlocks.

6.2.4 Performance Evaluation

In this Section, I will evaluate the performance of the various protocols generated and demonstrate the tradeoff the NetBlocks compiler offers in terms of performance and features. I will also evaluate the latency of communication for a simple Echo application and a real-world unmodified NGINX web server. I will also demonstrate the performance tradeoff when the network protocol is deployed underwater. Finally, I will compare the packet header overheads for the various protocols and the lines of code they generate to get a sense of the resources they require at deployment.

Evaluation Methodology

Testbed: For latency-critical applications typically found inside data centers, the experiments are run on two servers with 4-core Intel Xeon Gold 5122 CPUs running at 3.6 GHz with 64 GB of main memory and 16.5MB L3 cache. Both servers are running Ubuntu 22.04. Each node is equipped and connected to each other with a 100 Gbps Mellanox MT27800 family ConnectX-5 NIC that offers microsecond round-trip times.

For the underwater robotics evaluation, our generated protocols are run with the DESERT [124] underwater simulator that is built on top of the NS2 network simulator [79]. DESERT simulates the low-bandwidth, high-latency, and other network conditions in the underwater acoustic medium. A 4800-bit/second channel is used, along with the default MAC protocol, to prevent collisions.

Comparison Configurations: The primary aim of this evaluation is to demonstrate the feature-vs-performance tradeoff offered by the NetBlocks generated code. I will compare the performance of various protocols generated with NetBlocks with increasing degrees of features. The set of protocol configurations is listed below -

- **UDP-like**: Closely resembles the UDP protocol running on top of IP+Ethernet and does not have any features like reliability, in-order delivery, or signaling.
- **UDP-over-Ethernet**: Has the same features as UDP-like, but removes the IP layer and thus does not support routing. Routing is not needed for many deployments, like in an all-to-all topology or a deployment like underwater robots.

- **Inorder**: Builds on top of UDP-over-ethernet and adds a basic inorder delivery.
- **Reliable**: Builds on top of Inorder and adds reliable packet delivery with acks.
- **Signalling**: Builds on top of Inorder and adds signaling packets at connection establishment.
- **FullChecksumming**: Builds on top of Inorder and adds full packet checksumming.
- **ShrunkFields**: Similar to UDP-over-Ethernet but uses restricted fields.
- **Linux (UDP/TCP)**: A comparison of the above implementations is also done against the default Linux implementation to show that these implementations are competitive with existing implementations. UDP is used for the echo application and TCP for the NGINX application.

Protocol Header and Code Size Overhead

Before we evaluate the performance of protocols, I will first present the header sizes for the implementations and the size of the generated code. Minimizing protocol header sizes is critical for bandwidth-constrained applications, while reducing code size and memory footprint is important for memory-constrained deployments like IoT. Table 6.7 shows all the configurations and the header sizes along with their generated code size. We notice that the UDP-over-Ethernet protocol eliminates redundant headers and reduces the header size by over 50%. Other protocols that build over UDP-over-Ethernet add a small overhead for the specific feature-related fields they add. For example, In-order delivery adds a 32-bit field to store the sequence numbers, while Reliable delivery requires another 32-bit field for storing the acknowledgment sequence numbers. ShrunkFields shows the smallest header possible with 4 bits each for source and destination host identifiers (MAC addresses), 4 bits for source and destination app identifiers (port numbers), and 16 bits for the length field. For each of the configurations, similar to the header size, the lines of code required to implement are marginally more than the base case. The lines of code shown here are just the generated lines of C code and do not include the linked runtime library.

Applications

In this Section, I will compare the latency of the protocol configurations for a simple Echo application, an NGINX web server [41], and an underwater robotics simulation.

Echo Application: A simple echo application was implemented that ping-pongs messages back and forth between server and client and measures round-trip latency for each message. Since the server and the client do not perform work other than networking, this application allows us to isolate the overheads of each feature. To test the effects of the features like signaling, each message is sent over a newly established connection. The performance of protocols that don't use signaling is unaffected, other than the small local setup cost, since they don't send any messages. This application is evaluated on two hosts connected with a 100Gbps connection. The NetBlocks generated code is linked against a kernel bypass runtime to avoid syscall overheads. The default Linux (UDP) protocol is evaluated using the kernel implementation and suffers from the syscall overhead. For this evaluation, messages of 256 bytes are sent back and forth to thoroughly test the overhead from the schemes that involve checksumming.

Protocol Configuration	Header size	Generated code size
UDP-Like	42	252
UDP-over-Ethernet	20	241
Inorder	24	296
Reliable	28	364
Signalling	26	331
FullChecksumming	28	358
ShrunkFields	4	253
Linux (UDP)	42	–
Linux (TCP)	> 56	–

Table 6.7: Header sizes in bytes and the code size in lines of C code. All generated protocols are linked against a runtime library of 493 LoC.

NetBlocks Component	Lines of Code
Identifier Module	343
Inorder Module	177
Reliable Module	164
Routing Module	124
Signalling Module	131
Checksumming Module	123
Payload Module	85
Network Module	61
Framework	1,113
Runtime	1,826

Table 6.8: Implementation complexity of different components of NetBlocks. Each module is only a few hundred lines of C code.

Figure 6.12 shows the Cumulative Distribution Function (CDF) plot of the latency over 10,000 packets. We notice that the minimum latency is obtained with the UDP-over-Ethernet protocol configuration with a median latency of 3.25µs since it does not have any features like routing, checksumming, signaling, or reliable delivery. This is close to the single-digit microsecond latency possible with RDMA for packets of this size [99]. As we enable these features, the latency increases with signaling having the highest median latency of 6.0µs. This latency is almost twice the latency because for every connection, signaling packets have to be exchanged before the actual messages are sent. For the protocol that implements reliability, an acknowledgment needs to be sent for every packet, but the sender doesn't have to wait for the acknowledgment before sending the next message, and hence, this scheme increases the latency slightly. Finally, we see that despite having the smallest header size, the ShrunkFields configuration has more latency overhead than the UDP-over-Ethernet protocol. Despite having the same features, this configuration needs to generate bit-packing code with masks and shifts, adding latency in the host processing. With this observation, we can conclude that the different schemes provide a real tradeoff in different metrics. The Linux configuration is an order of magnitude slower than our slowest scheme because it doesn't use a kernel bypass stack and suffers from the overheads of syscalls and interrupts.

This evaluation demonstrates that, unlike the all-or-nothing approach of UDP and TCP, our approach of generating protocols with select features provides a spectrum of performance. This allows the users to have a pay-for-what-you-use policy when it comes to features. The DSL inputs for each of these protocols are less than 50 lines of C++ code, allowing the user to switch between vastly different protocols with minimal effort.

NGINX: For the next evaluation, we use a real-world web server NGINX [41] to demonstrate that NetBlocks generated protocols are all supported by applications written for the POSIX API. An unmodified NGINX web server written with TCP sockets is run with the 7 protocol configurations generated from NetBlocks. GET requests are sent that download static files off the server. The files are stored in an in-memory file system to avoid variance from disk reads. The round-trip latency is measured over 10,000 GET requests downloading a file of 850 bytes. The total payload also

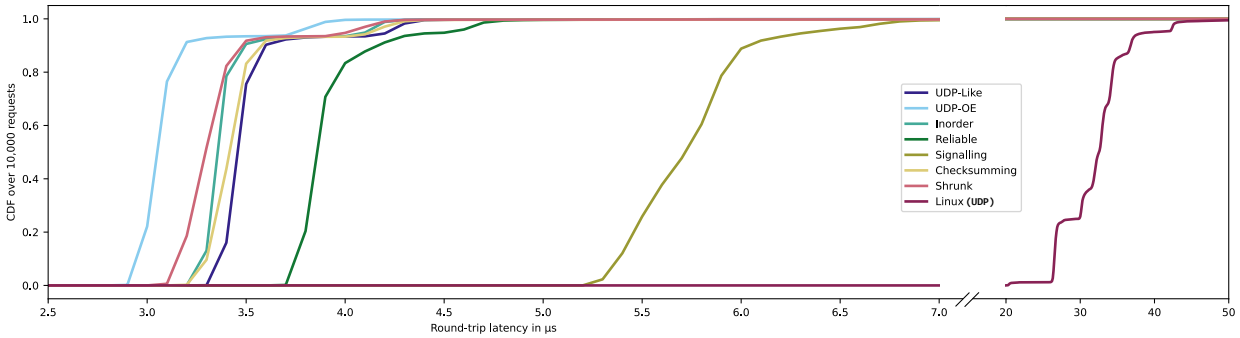


Figure 6.12: CDF of the round-trip latency over 10,000 messages for the Echo application with the 8 protocol configurations. The message size used for the ping-pong messages is 256 bytes.

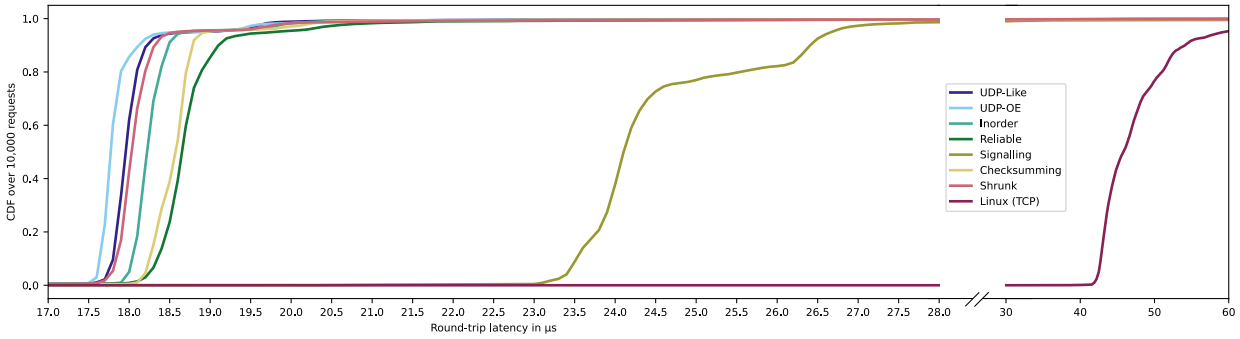


Figure 6.13: CDF of the round-trip latency over 10,000 requests for the Nginx application with the 7 configurations.

includes the HTTP response headers along with the file.

Figure 6.13 shows the CDF plot for the 10,000 requests. We can observe a similar trend to the Echo application, even though the absolute latency numbers are higher. This is due to the fact that the server has to perform system calls to read the files. However, we demonstrate that the benefits of specializing the protocol are translated to real-world applications. In this evaluation, the overhead of Checksumming is significantly higher than that of the Echo Application because the payload size is much larger. Similarly, the overhead of Reliability is higher since the NGINX server sends the response over multiple messages, each requiring its own acknowledgment. Thanks to NetBlocks's design of exposing the same API to the application regardless of the generated protocol, we are also able to run NGINX with a stateless UDP-like protocol without any source modifications despite being originally written for TCP. To our knowledge, this is the first time NGINX has been run with a header as small as 4 bytes. For the comparison Linux implementation, we use TCP because the two protocols are not compatible in Linux.

Underwater Robotics: For the next evaluation, we run the NetBlocks generated stacks with the DESERT [124] underwater simulator to evaluate the effect of feature selection on throughput in a low-bandwidth environment. Since the latency of the communication is bottlenecked by the acoustic medium, the host-side overhead is negligible, and our techniques don't affect latency. However, being able to compactly pack the headers helps us better utilize the low-bandwidth channel. For our evaluation, a simulation is set up between two hosts where one host repeatedly sends a 16-byte

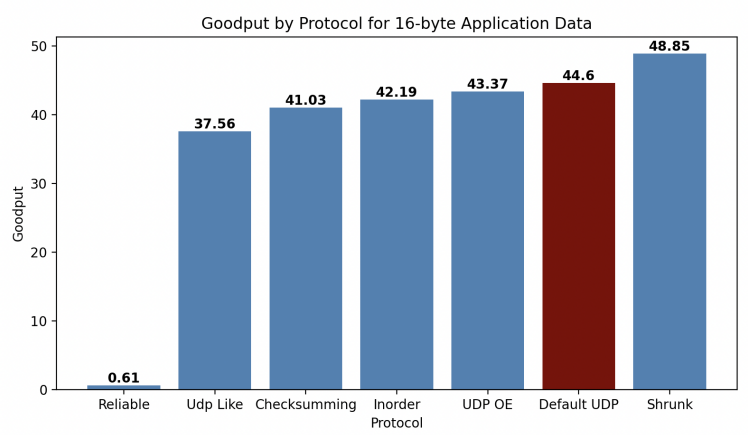


Figure 6.14: Goodput measurement for the underwater robotics sensor simulations with DESERT. The payload size is 16 bytes.

payload to the other host over a 4800 bps channel. The 16-byte payload mimics sensor measurement readings on the robot. Figure 6.14 shows the measured Goodput [111] for our custom protocols and the default UDP implementation in DESERT. The goodput (i.e., the number of useful application bytes transmitted per second without considering retransmissions) for the Shrunk scheme is the highest. By minimizing the header sizes, we can better utilize the channel for the actual payload. The default UDP implementation in DESERT has been optimized for the underwater scenario and is the second best. As we add more features, the goodput gradually decreases and is extremely low when reliability is enabled. This is because, with reliability, the receiving host also has to send packets for the acknowledgments, which adds contention to the channel, and the underlying MAC protocol cannot utilize the channel optimally. This further elaborates the point that we need custom protocol generators like NetBlocks. With just TCP and UDP, if the system requires in-order delivery, the operator is forced to use TCP and sacrifice the performance with the acknowledgments and handshakes.

Discussion

These three applications and scenarios show that when minimizing latency or maximizing throughput is of concern, creating a custom protocol with NetBlocks can have a real impact. It shows that each network feature comes with a price, and creating a custom protocol with only the salient features your application needs is much better than the two-sizes-fits-all approach with TCP and UDP. We also demonstrate that NetBlocks compiler generates highly optimized kernel bypass code; thus, the latency results are much better than the hand-tuned Linux implementations.

6.3 BREeze: a High-Performance Regular Expression Compiler

All the case studies I have described so far followed the embedded DSL model, where the DSL itself is implemented as a library in the host language C++ using BuildIt, and the end-user writes a full program with this library, where the output of the execution then generates an optimized version for that program. For the next case study, I will present a different model where I will show that an

interpreter for a single language, like Regular Expressions (Regex), can be turned into a compiler by exploiting the Futamura projections [69]. This work was done jointly with a master's student and an undergraduate student in the group that I mentored. Although I was involved with the design and implementation, all the implementation was done by Tamara Mitrovska as part of her master's thesis [129] and another undergraduate student, Alice Chen.

Let us start by understanding the domain of Regex matching before getting to our compiler design and implementation. Regular Expression matching is a very old problem with ubiquitous applications in software engineering. Due to the wide range of regular expressions, one general implementation approach does not always yield the best performance for matching all types of patterns. For example, regex libraries implemented with simple backtracking do not perform well for expressions with high ambiguity [53]. As a result, numerous techniques throughout the years attempting to solve the problem in both time and space efficient manner build upon the same underlying principle that regular expressions can be represented as finite automata [188, 68, 9]. Naturally, since it is difficult to find one implementation that best suits all applications, a compiler-based solution that takes the entire regular expression into account to decide the best approach to perform the matching is required to generate different implementations and explore the tradeoffs.

In this case study, we will first implement a simple code generator for a given regular expression by writing an interpreter for Regex that accepts a Regex and the string to match, and then partially evaluate it for the Regex string to generate a program that just accepts the input string and matches the specific regular expression. The staged program P_S thus behaves like a compiler, since it accepts a Regex and generates specialized code to match that Regex. Such conversion of interpreters to compilers using staging or partial evaluation is called the First Futamura Projection [69]. In the next section, we will start by implementing a simple Regex compiler and then optimize the generated code by introducing a scheduling language that allows handling non-determinism at various sites in different ways.

6.3.1 Implementing a Basic Regex Compiler

A simple Regex matching interpreter is a function that accepts two inputs: a string representing the regular expression and the string to match it on. This function then does the matching and returns if there is a match for the Regex in the input string. A more complex version can return the number of matches, the locations of matches, and even the substring that actually matched the Regex. We will start our implementation with the basic variant and slowly add more features. The simplest way to implement a Regex matcher is by writing a Regex interpreter. Matching a Regex is inherently a non-deterministic process due to the existence of the `*`, `+`, `?` operators. In fact, most Regex can be thought of as a simple NFA where each character in the Regex is a state of the NFA, and what characters need to be matched to progress along the regex represent the edges of the NFA. Such a conversion of a Regex to an NFA is referred to as Thompson's Construction [177]. There are several ways to evaluate an NFA on an input, one of which is to simulate the evaluation of an NFA on an input in a deterministic way. This can be done by either explicitly converting the NFA to a DFA and then simulating a deterministic automaton, or directly simulating an NFA by simply keeping track of all active states of the NFA in parallel as the matching continues. The latter approach is easy when implementing an interpreter since it doesn't require any preprocessing and can be simply implemented with a doubly nested loop - one that goes over the input string and the other that goes over each non-special character in the Regex. Two state vectors are maintained to keep track of

states that are currently active. The current state vector is initialized to just the initial character being active, and during each iteration, the next state vector is produced based on the current active states, whether the current character can be matched with that state, and the state it would transition to. At this point, the state vectors are swapped, and the execution is continued. This continued till all the characters in the input state are exhausted. If at the end the end state of the Regex is active, the Regex is said to have matched the string. This basic algorithm can be further modified to keep track of the matches, the count, and so on.
This naive algorithm, however, is extremely slow since at each character it has to match every state regardless of the fact that only a subset are active. The vectors themselves also need to be updated and swapped around, which can be expensive. However, if we use BuildIt's multi-staging, we can partially evaluate away a lot of this computation. To apply the Futamura projections, we would require the Regex itself to be available in the *static* stage. All the computation on the characters in the Regex, including the state vectors updates, can be performed in the *static* stage. Even the iteration over the state vector to test if a state is active can be partially evaluated away, generating code that matches only the states that are active. The input string and all the computations depending on it, specifically the iteration over the string and the equality comparison of each character against the character in the Regex, are performed in the *dynamic* stage. Thus, by applying this partial evaluation, a simple code with just a sequence of conditions and no variables or updates was generated. Figure 6.15 and Figure 6.16 show the entire implementation of such a Regex matcher written using BuildIt's `static_var<T>` and `dyn_var<T>` types.

There are two main components of this implementation, the `progress` function and the `match_regex` function. The `match_regex` function is the entry point for the matching algorithm and accepts a Regex as *static* parameter and the string to match as a *dynamic* parameter. It then runs the whole matching process through the two-level nested loop. Notice that the loop that goes over the character in the input string is written with a *dynamic* index while the one that goes over the characters in the state vector is written using a *static* iterator. There are few optimizations here like `early_break` which ensures that spurious nested comparisons of the form `if (x == 'a') { if (x == 'b') {...}}` are avoided. Other than that, this is a pretty basic implementation of the Regex matcher turned into a compiler. The `progress` function shown in Figure 6.15 is a helper function that executes entirely in the first stage and updates the active state in the next vector given a match in the current vector. Even though this is a complex function with recursions, it is completely evaluated in the *static* stage to produce no code.

The most noteworthy part of this implementation is that this code has the potential side-effect leak problem described in Chapter 2 at Line 20. Notice that the line performs a condition on a *dynamic* variable `str[to_match] == m` and within the branch calls `progress` that updates the `static` variables. If the side-effect leak problem wasn't handled properly, this would lead to the generation of wrong code since the updates to the `next` array in `progress` would leak to the else branch. Since BuildIt implements **REMS**, which is designed to solve this problem, this implementation instead generates code with nested if conditions and nested loops based on the Regex character.

Let us look at the generated code when this function is invoked with a simple regex like `ab*c`. Figure 6.17 shows the generated code. As we can notice, the generated code has no state vector arrays or updates to any variables, but is just a ladder of if-then-else statements. The only update is to index into the string, which keeps getting updated. Since BuildIt is invoked with the `feature_unstructured=true` option, the generated code has if-then-else and gotos instead of while loops. However, we can notice one rudimentary loop corresponding to the matching of `b*` in the

```
#include <iostream>
#include <cstring>
#include "builder/dyn_var.h"
#include "builder/static_var.h"
#include "blocks/c_code_generator.h"

dyn_var<int (char*)> d_strlen = builder::with_name("strlen");

dyn_var<int> match_regex(const char* re, dyn_var<char*> str);
bool is_normal(char m) {
  return m >= 'a' && m <= 'z'  m >= 'A'
    && m <= 'Z'  m >= '0' && m <= '9';
}
void progress(const char *re, static_var<char> *next, int p) {
  int ns = p + 1;
  if (strlen(re) == ns) {
    next[ns] = true;
  } else if (is_normal(re[ns])  '.' == re[ns]) {
    next[ns] = true;
    if ('*' == re[ns+1]) {
      // We are allowed to skip this
      // so just progress again
      progress(re, next, ns+1);
    }
  } else if ('*' == re[ns]) {
    next[p] = true;
    progress(re, next, ns);
  }
}
int main(int argc, char* argv[]) {
  builder::builder_context context;
  context.feature_unstructured = true;
  auto ast = context.extract_function_ast(match_regex, "match_re", argv[1]);
  std::cout << "#include <string.h>" << std::endl;
  block::c_code_generator::generate_code(ast, std::cout);
  return 0;
}
```

Figure 6.15: The main function and the progress function from the simple Regex compiler implementation

string.

Thus, the simple Regex compiler we implemented generates straight-line code for Regex. The actual implementation in BREeze is longer with more special cases and support for conditionally matching sub-strings, or gathering the actual matches, and so on. The full BREeze compiler also implements support for groupings and special characters. Table 6.9 shows all the expressions that are supported by the breeze compiler and their description.

However, as we noted before, just one strategy to handle non-determinism isn't enough. The above implementation handles non-determinism by simultaneously matching all the possibilities.

```
dyn_var<int> match_regex(const char* re, dyn_var<char*> str) {
  // allocate two state vectors
  const int re_len = strlen(re);
  static_var<char> *current = new static_var<char>[re_len + 1];
  static_var<char> *next = new static_var<char>[re_len + 1];
  for (static_var<int> i = 0; i < re_len + 1; i++)
    current[i] = next[i] = 0;
  progress(re, current, -1);

  dyn_var<int> str_len = d_strlen(str);
  dyn_var<int> to_match = 0;
  while (to_match < str_len) {
    static_var<int> early_break = -1;
    for (static_var<int> state = 0; state < re_len; ++state)
      if (current[state]) {
        static_var<char> m = re[state];
        if (is_normal(m)) {
          if (-1 == early_break) {
            // Normal character
            if (str[to_match] == m) {
              progress(re, next, state);
              // If a match happens, it cannot match anything else
              // Setting early break avoids unnecessary checks
              early_break = m;
            }
          } else if (early_break == m) {
            // The comparison has been done already, let us not repeat
            progress(re, next, state);
          }
        } else if ('.' == m) {
          progress(re, next, state);
        } else {
          printf("Invalid Character(%c)\n", (char)m);
          return false;
        }
      }
    // Now swap the states and clear next
    static_var<int> count = 0;
    for (static_var<int> i = 0; i < re_len + 1; i++) {
      current[i] = next[i]; next[i] = false;
      if (current[i]) count++;
    }
    if (count == 0)
      return false;
    to_match = to_match + 1;
  }
  // Now that the string is done, we should have an end in the state
  return (char)current[re_len];
}
```

Figure 6.16: The match_regex function from the simple Regex compiler implementation that accepts a regex as a *static* parameter and the string to match as a *dynamic* parameter.

```
int match_re (char* arg1) {
  char* var0 = arg1;
  int var1 = strlen(var0);
  int var2 = 0;
  if (var2 < var1)
    if (var0[var2] == 'a' {
      var2 = var2 + 1;
      label0:
      if (var2 < var1) {
        if (var0[var2] == 'b') {
          var2 = var2 + 1;
          goto label0;
        }
        if (var0[var2] == 'c') {
          var2 = var2 + 1;
          if (var2 < var1)
            return 0;
          else
            return 1;
        } else
          return 0;
      } else
        return 0;
    } else
      return 0;
  else
    return 0;
}
```

Figure 6.17: Code generated from the Regex compiler for the input ab*c. Notice that this generated code effectively only has one loop.

Expression	Description
`.`	any character except newline
`x?`	zero or one x
`x+`	one or more x
`x*`	zero or more x
`(x|y)`	x or y
`[xyz]`	character class
`[^xyz]`	negated character class
`[a-z]`	character range
`[^a-z]`	negated character range
`x{n}`	x repeated n times
`x{n,}`	x repeated n or more times
`x{n,m}`	x repeated between n and m times inclusive
`\d,\w,\s,\D,\W,\S`	special character classes

Table 6.9: List of different expression and their descriptions supported by the BREeze compiler.

For some Regex with lots of ambiguity, this blows up and code upwards of 100s of thousands of lines is produced, which, even though it is correct, is not the most efficient. Furthermore, Regex also presents the ability to match strings in parallel by splitting the string into chunks and matching them to different parts of the Regex. The exact strategy for each needs to be chosen. To support this, implement a scheduling language in BREeze that lets the user make these choices. I will discuss the scheduling options that help improve performance next.

6.3.2 Scheduling Regex matching in BREeze

In this section, I will go over the various scheduling options BREeze provides and the benefits they offer. All the scheduling options are provided as a second input string along with the Regex of the same, with one character for each specifying the scheduling decision for the location in the Regex.

Splitting the Regex

This option, denoted by `'s'`, combines the basic parallel matching approach with backtracking. More specifically, patterns with alternations can be compiled such that each alternation part is matched separately. This resembles backtracking because if one alternative option fails, we have to go back and try another one. To demonstrate how this works, consider the Regex `(Tom.{10,15}river|river.{10,15}Tom)`. For this expression, the normal approach generates one function that performs parallel matching. Instead, we can generate 2 functions `f1` and `f2` for matching `Tom.{10,15}river` and `river.{10,15}Tom` respectively. Finally, to get a match, we run `f1` first, and if it fails, we run `f2`. To denote a split like this, we pass the following string as the flags option in `regex_match`. `.s.............s.....` The flags string is the same length as the regex. To denote that we want to split the expression at index `i`, we set `flags[i] = 's'`. If there are no special options for index `j`, we just set `flags[j] = '.'`. When the user-provided schedule gets parsed into a Schedule struct, the split option is set to true if there are any `s` characters in flags. Although the split option was inspired by alternation expressions, it works in general for splitting at any position in the pattern, even with no alternations. Compiling `(Tom.{10,15}river|river.{10,15}Tom)` with this option takes only 70 milliseconds (compared to more than 5 minutes without any scheduling). The generated code has 3 functions: 1 function for each of the alternation options (we will call these functions helpers) and 1 main function that calls the helper functions. Only the main function implements the partial match logic. The helper functions implement an anchored match at a specific position in the string passed as an argument from the main function. We achieve this by keeping a working set of all the functions that need to be generated during the code generation stage. BuildIt first starts generating the main function. When the main function calls another dynamic function, we add that function to the working set and BuildIt generates it once it is done with the previous function. The number of active states at any time during the generation of each of these functions is generally lower than in the basic approach because the work is distributed across multiple functions. This results in generating more compact code, which improves both the compilation and running times.

Matching multiple characters at once

In the normal approach, literal strings that appear as part of a regex, such as Tom, are matched one character at a time. As mentioned before, this results in a lot of states being active at the same time, which slows down the code generation process. One reason for this is that each character results in a separate if condition in the generated code. Combined with different paths the code can take based on the outcome of the if statements, the matching code can become very long and complicated. To decrease the total number of active states and simplify the code generation, we introduce the join option to compare multiple consecutive characters at once with `memcmp`. For example, with this option, the string Tom is represented with only one state instead of 3. Similarly, the generated code has only one if condition instead of 3. This option not only simplifies the compilation process but also has some benefits in terms of running times. Namely, when compiling the generated code, the backend C compiler can represent `memcmp` as a single instruction, which adds an extra optimization of being able to compare multiple bytes with a single instruction call. In the normal approach, one instruction would compare only a single byte. The join option can be specified as part of the flags string in the schedule, just like the split option using the character `'j'`.

Interleaving

As mentioned before, the match code is generated by activating the first state of the regex for each position in the dynamic string that we are matching. This is the main reason behind having many states being active at the same time. To control the frequency at which we activate new states to start matching from the beginning of a pattern, we introduce the interleaving option. The user can specify the interleaving frequency with the `interleaving_parts` option in `RegexOptions` which later gets copied to the `Schedule` struct. Let n be the number of interleaving parts. Interleaving works as follows. There are n different functions generated, such that the i^{th} The function is responsible for matching the pattern starting at every position p in the dynamic string where $p \bmod n = i$ These functions are independent of each other because they operate from different starting positions in the dynamic string. Therefore, we can run them in parallel. We parallelize the code by adding `#pragma omp parallel for` to the for loop that calls each of the n functions using the annotations discussed in Chapter 5.

This option lets us compile `(Tom.{10,15}river|river.{10,15}Tom)` with 16 interleaving parts in around 8 seconds. The compilation time can be improved by varying the number of interleaving parts or combining these options with the split and join options.

Splitting the String

Although interleaving helps with the compilation times, it does not, in general, improve the running times. The main reason for this is that each interleaving function still has to traverse the entire string to find a match.

To avoid iterating over the entire dynamic string serially, we split it into blocks that can be traversed in parallel. The user can specify the block size with the `block_size` option in `RegexOptions` which gets copied into the `Schedule` struct. For the rest of the discussion, we refer to this scheduling option as the block option. The generated code with this option consists only of one function, assuming we are not using any of the other scheduling options from above. Among other arguments, this function takes in an index s in the dynamic string that we need to start matching from. Let L

be the length of the dynamic string and B the block_size specified by the user. Then, the number of blocks is $N = \lceil \frac{N}{B} \rceil$. We simulate splitting the string into N blocks by calling the generated function N times such that in the i^{th} call we pass $s = B * i$ for the string start position. Internally, the generated function continues looking for partial matches only until their starting positions correspond to a position in the dynamic string p such that $s \leq p < s + B$. The N function calls can be run in parallel. The block option can be used in combination with the other scheduling options described above. Generally, it has better compilation times compared to interleaving. We can compile `(Tom.{10,15}river|river.{10,15}Tom)` with the join and block options together in around 42 milliseconds. Moreover, block significantly improves the running times as seen in the next section.

6.3.3 Evaluations

As with other case studies, one of the main goals of this case study is to minimize the amount of effort required to generate highly performant code. We will thus compare both the lines of code and the performance of the generated code against state-of-the-art frameworks.

Implementation Effort

Implementing BREeze on top of BuildIt allowed us to generate highly specialized code with very simple and concise implementation. To demonstrate this, we compare the number of lines of code in our code-base to the total lines of code in Hyperscan [188], RE2 [48], and PCRE2 [144]. For fairness, we only count the lines of code inside the `src` directory of the official GitHub repositories of each of these libraries, excluding any testing or timing code. Table 6.10 shows the comparison of the lines of code for the different frameworks. The implementation of BREeze is about 120 times smaller than the largest framework, Hyperscan, and about 17 times smaller than the next smallest framework, RE2. This shows a significant reduction in lines of code and thus the effort required to implement the frameworks.

Framework	Lines of Code
BREeze	1,564
RE2	26,587
Hyperscan	187,033
PCRE2	133,995

Table 6.10: Number of lines of source code used for implementation of each of the libraries. The count excludes the code used for testing and benchmarking.

Performance Evaluation

In this section, we analyze the compilation and running times for finding a single partial match in a long string. We ran the experiments on an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz machine with 128GB memory, 48 cores, and 2 threads per core. We used the teakettle_2500 and snort_literals regular expression sets with the Gutenberg and Alexa200 texts, respectively, from a Hyperscan performance analysis blog [182]. Additionally, we used the Twain benchmark [145] to

show our performance on a small set of regular expressions compiled with hand-optimized schedules. Table 6.11 shows the execution time for the matching of the different frameworks as compared to BREeze. For BREeze, we manually tried different schedules and picked the ones that compiled in a reasonable amount of time and ran the fastest.

regular expression	BREeze	RE2	HScan	PCRE2
`Twain`	**0.0001**	0.0006	0.0020	0.0005
`(?i)Twain`	**0.0001**	0.0004	0.0020	0.0004
`[a-z]shing`	**0.0022**	0.0051	0.0025	0.0529
`(Huck[a-zA-Z]+|Saw[a-zA-Z]+)`	1.539	3.999	**0.623**	2.052
`3[a-q][^u-z]1x`	0.131	0.066	**0.011**	0.110
`(Tom|Sawyer|Huckleberry|Finn)`	0.027	0.035	**0.003**	0.160
`(?i)(Tom|Sawyer|Huckleberry|Finn)`	0.015	0.012	**0.002**	0.160
`.{0,2}(Tom|Sawyer|Huckleberry|Finn)`	0.041	0.045	**0.003**	3.734
`.{2,4}(Tom|Sawyer|Huckleberry|Finn)`	0.041	0.035	**0.003**	3.594
`(Tom.{10,25}river|river.{10,25}Tom)`	5.195	13.349	**0.815**	26.962
`[a-zA-Z]+ing`	**0.0015**	0.0029	0.0025	0.0536
`\s[a-zA-Z]{0,12}ing\s`	0.011	0.0048	**0.0047**	0.0362
`([A-Za-z]awyer|[A-Za-z]inn)\s`	5.796	2.964	**0.336**	87.339
`["'][^"']{0,30}[?!\.]["']`	**0.027**	0.030	0.034	0.058

Table 6.11: Performance comparison of regular expression frameworks.

From the table, we observe that BREeze is consistently faster than PCRE2, and faster than RE2 for most of the expressions. Although we are doing better for some expressions, Hyperscan in general performs the best for this benchmark. This is mainly because of the wider range of schedules supported by BREeze as compared to PCRE2. Hyperscan is a heavily optimized and hand-vectorized regex matcher optimized for regular expressions and implements a lot more optimizations. Future work would require implementing those optimizations as extensions to BREeze to further boost the performance.
Overall, we can conclude that a staged interpreter for a language like Regex written on top of BuildIt can provide comparable if not better performance while reducing the effort by up to 2 orders of magnitude.

6.4 Other Applications of BuildIt

Besides the case-studies described above, BuildIt is being actively used in projects in and outside the group. These projects are applying BuildIt's multi-staging capabilities to domains like robotics and array processing, even targeting novel architectures like FPGAs.

6.4.1 MARCH: Multi-staging ARray Compiler for High-Level Synthesis

MARCH [142] is an Array DSL built on top of BuildIt for targeting FPGAs using High-Level Synthesis. MARCH provides a high-level DSL for programming with arrays and index notations.

MARCH uses BuildIt in a 3-stage execution mode where a first pass of BuildIt is used to extract the expressions and control flow. The extracted code represented using ASTs from the `block` namespace is processed to create and assign FPGA-specific scheduling, which is inserted into the code as opaque function calls and annotations in the first stage code. This AST is then again interpreted in a second execution of BuildIt using the `dyn_var<T>` and `static_var<T>` types in the visitors for the AST. This lowered code is then compiled through a special backend, a modified version of the C Code Generator, to generate Vitis HLS code to run on FPGAs. In comparison with state-of-the-art ADLs, MARCH obtains a geometric mean speedup of 1.07x on the Polybench benchmarks. Moreover, MARCH shows how multi-staging inherently offers better ways to build parametric HLS accelerator generators by implementing the Faber mutual information accelerator, originally implemented by mixing hand-written HLS with Python generators. The BuildIt based MARCH array language implements it in 2x fewer lines of code than the original implementation.

6.4.2 Code Generation for Robotics Path Finding using BuildIt

A work in progress project [74] is focused on optimizing path-finding algorithms for robotics applications. Linear algebra kernels that reason about the physical properties of robots and their environments lie at the heart of robotics. The path-finding algorithms have to solve expensive kernels based on inputs about the environment, robot topology, and other constraints. In many cases, the constraints of the environment or the robot are known at compile time as domain information. Oftentimes, roboticists exploit this information to hand-optimize these kernels. However, this makes the process very time-consuming and brittle. This group is using BuildIt to partially evaluate these path-finding algorithms to tailor these expensive computations to specific scenarios automatically, to be able to optimize the code using fusion, interleaving, and vectorization. The specific project partially evaluates the linear algebra library Eigen [81], which is the backbone of these computations. The system is able to obtain multiple orders of magnitude improvement in the latency of the computations. This work is still in progress and will be submitted soon.

Besides these, the BuildIt framework was used in the COMMIT group to build two DSLs. SHIM [148] is a domain-specific language for the UNITE abstraction with application to high-performance compression algorithms. Another project implemented the StreamIt [176] DSL on top of BuildIt for targeting CPUs and a novel reconfigurable data-flow architecture. The new StreamIt compiler is being used to implement streaming applications from the signal processing and machine learning domain [60, 59].
Separately, another project, Dynamic-By-Default [109] is currently being developed that automatically converts existing C code into BuildIt code with `dyn_var<T>` types on all variables and expressions to extract the code as it is. This will allow for easy migration of code bases into BuildIt, streamlining future adoption.

Chapter 7

Extensible and Contextual Debugging for DSLs

> "As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."
>
> Maurice Wilkes

Domain-specific languages and Compilers are a key tool in the efforts for building high-performance software. DSLs combine the productivity benefits of writing entire applications in a high-level abstraction with the performance benefits of the generated low-level specialized code. The techniques described in Chapter 6 and the use of multi-staging in a seamless system like BuildIt also greatly improve the productivity for the DSL designer by implementing operator fusion, data-flow analyses, static and data-dependent scheduling, data structure and data layout optimizations, and even partially evaluating simple interpreters and converting them into compilers in the process. These techniques allow domain experts to build language abstractions that generate state-of-the-art code with a fraction of effort.

But writing code and running it is not the only part of the developer experience that contributes to their productivity. No matter what language abstractions are chosen, programmers will eventually make mistakes and inadvertently introduce bugs. This, however, is not the end of the world since language and toolchain developers have spent years building tools and techniques to find and remove these bugs. This process of isolating bugs and removing them is called debugging. The tools that help with this process generally fit in the broad category of tools called debuggers. Debuggers come in a wide range of complexity and techniques from inserting print statements in the program and passively observing the behavior at runtime, to sophisticated techniques like time-travel debugging [13, 183, 116, 64, 154] where the developer uses specialized tools to stop, step-through or even rewind the execution of the program at runtime and observe the behavior to

better understand the bugs. As a lot of popular languages like C, C++, CUDA, Python, JavaScript have evolved over the ages, their debuggers have also become more and more sophisticated over time, like gdb/lldb/WinDbg [161, 143, 128] for C, C++ and other native code, pdb [67] for Python, cuda-gdb [47] for NVIDIA GPUs and so on.

The interaction of debugging facilities and domain-specific languages is, however, an interesting case. Firstly, since domain-specific languages by definition are focused on a certain set of applications, the community of users and the number of experts building compilers and other toolchains in the area are far smaller than those of mainstream general-purpose languages. Secondly, since the compilers for these domains are also complex domain-dependent systems, the bugs could be introduced either due to the mistake of the end-user or due to some mistake during the compiler and other runtime development, making it even harder to isolate bugs. Finally, existing debuggers also fall short when supporting DSLs. In the DSL setting, the end-users write their programs in a higher-level DSL, which then goes through the compiler and produces low-level C, C++, CUDA, or other output depending on the target platforms. In some cases, DSLs can also be chained, where very high-level declarative DSLs produce inputs for intermediate DSLs, which then generate the low-level code. Due to this disconnect in the code that the developer wrote, being very different from the code that actually runs, even attaching a difference gives little to no visibility into the bugs. Existing debuggers and debugging formats are also painfully difficult to modify, given their complexity and size, and due to the various domain-specific debugging needs of each DSL. Some DSLs may not even have printing facilities at a fine-grained level, taking away the last hope for the developers, leaving them with only the option to stare at their implementations, hoping to identify the bug.

To address this disconnect between the general-purpose programming world and the DSL world, I will introduce D2X [21], which stands for **D**ebuggers for **D**SLs that is E**X**tensible and Conte**X**tual (pronounced as Detox). D2X is a C++ library that works with the implementation of your DSL compilers and turns ordinary unmodified general-purpose debuggers like gdb and lldb into powerful domain-specific debuggers that can access and interact with information in the context of the DSL. Furthermore, D2X APIs are extremely simple and allow for arbitrary extensions to the debugging toolchain with little to no changes. I will demonstrate the capabilities of D2X by applying it to a traditional compiler for the graph domain language GraphIt to debug the generated CPU parallel code while requiring about 1.4% changes to the code base. I will also demonstrate applying D2X to BuildIt itself to provide debugging support for any DSL built with it automatically without any developer effort, further simplifying the DSL design workflow for non-compiler experts. I will thus make the case that D2X increases the productivity of not only the end-user but also current and future DSL designers.

7.1 Challenges to DSL Debugging

I have already hinted above at the ways in which the debugging experience for DSL remains significantly more challenging than general-purpose languages. I will now summarize the challenges down to three main core issues: i) Disconnect in DSL Source and Generated Code, ii) Use of Complex Data Structures in DSLs, and iii) the fact that Existing Debugging Tools are very Rigid. Let's discuss each of these in detail to better appreciate the solution D2X provides in the upcoming

section.

7.1.1 Disconnect in DSL Source and Generated Code

As seen in Chapter 6, domain-specific language compilers typically parse some input provided in a high-level language and generate low-level C, C++, or CUDA code. This approach is taken by DSLs like GraphIt [201, 199, 26], SIMIT [106] and Diderot [36]. Other DSLs like Halide [146], Taichi [88], and Tiramisu [7] embed themselves into host languages like Python or C++ for providing a familiar interface. Some DSLs like TACO [105] sometimes provide both these interfaces. Even though the DSLs built using BuildIt's proposed methodology don't need to implement a parser, they still provide a higher-level interface to the user. The input code is passed through a series of analysis and transformation passes and combined with scheduling inputs before generating the final code. As we have seen often, the code is parallelized, vectorized, or moved to a GPU as specialized kernels to get the best performance. Naturally, the generated code has very little resemblance to the input code. For example, in the popular graph DSL GraphIt, we have previously seen that a single call to an `edgeset.apply` operator often produces 100s of lines of low-level code. This is done to implement various optimizations such as different iterations and parallelization strategies, hybrid scheduling, handling different data structures, and graph blocking. An end-user attaching a debugger to the generated code would have a very hard time trying to find correspondence between the generated code and the input code. This problem is further exacerbated by the fact that a single function can be compiled in different ways depending on the calling context.

Figure 7.1 shows a User Defined Function (UDF) (Line 1) written in GraphIt to be applied to each edge in an edgeset. The same UDF is used by two different calls to `edgeset.apply` operators (Line 5- 6). Depending on the schedule applied to these operators, the UDF is compiled in very different ways. Figure 7.2 Line 2 and 5 show the two generated versions of the same UDF tailored for the respective call sites for correctness and performance. Similarly, in BuildIt style DSLs, each generated line of code is produced from not one line but an entire call stack. Most DSL compilers, especially those that have their own frontend, are able to obtain and keep track of source line numbers, but that is often not sufficient. The compilers have to present this information to the end-user in such a way that they can trace back faults and bugs to the exact input they wrote, despite all the complex domain-specific transformations. The end-user should also be able to insert breakpoints in the generated code based on source locations in the DSL input. The data format to capture this debug information and the support to access it in the debugger are simply not present. Just attaching a general debugger to the generated code is not enough.

7.1.2 Use of Complex Data Structures

DSLs targeted for specific domains require specific data structures from the domain. A single object in the DSL often maps to one or more complex objects in the generated code. For example, in the Sparse Tensor DSL TACO, individual sparse tensors are stored in one or more formats such as CSR, COO, or BCSR based on the operation being performed. Even though the data structure might be a single variable in the DSL abstraction, it might be broken down to several structures, maps, and arrays, or sometimes even complex to navigate pointer-based data structures. In the case of code generated for GPUs, a copy for the device and host is generated, and part of the data structure can reside on the host or the device. Once again, the end-user would have a hard time debugging the

```
func updateEdge(s: Vertex, d: Vertex)
   nrank[d] += orank[s]
end
func main()
    #s1# edges.apply(updateEdge) // PUSH Schedule
    #s2# edges.apply(updateEdge) // PULL Schedule
end
```

Figure 7.1: GraphIt algorithm input with the same User Defined Function (UDF) **updateEdge** used with two operators, one with PUSH schedule applied and the other with PULL.

```
void updateEdge_1(int s, int d) {
    atomicAdd(&nrank[d], orank[s]); // For #s1#
}
void updateEdge_2(int s, int d) {
    nrank[d] += orank[s];           // For #s2#
}
```

Figure 7.2: Generated code for the GraphIt input in Figure 7.1. The same UDF is generated into two separate versions suited for the two call sites.

state of these variables just by attaching a debugger to the generated code. The DSL compiler needs to encode how individual objects in the source code map to objects in the generated code and the logic to access their state from the debugger.

7.1.3 Debugging Tools are Rigid

Current debugging tools like GDB, LLDB, and WinDbg have various capabilities such as setting breakpoints, reading and writing variables and registers, displaying source information, and calling functions from the applications. While these are good for languages like C or C++, DSLs and DSL compilers have a lot of DSL-specific state often hidden inside the compiler. This includes results of the domain-specific passes, statically inferred types of variables in a dynamically typed language, among others, which are crucial for debugging the generated code. For example, Seq [155], the DSL for genomics, is a dynamically typed language similar in syntax to Python but generates high-performance native code after statically inferring types for all variables. This type of information is neither present in the source code nor in the generated binary. Displaying this type of information would require extending the debugger and the binary format used to encode the debugging information. BuildIt and DSLs built with BuildIt, like NetBlocks and BREeze, also have a similar problem where the state of the first-stage variables is not visible in the generated code but is critical for debugging.

There are two problems with extending the debugger to display this semantic information. i) The debugging information is stored in complex formats that are hard to understand and harder to extend. For example, DWARF [40], the popular debugging format used on Linux, has a standard that spans 459 pages, and the complexity required to understand it is beyond the scope of a typical domain

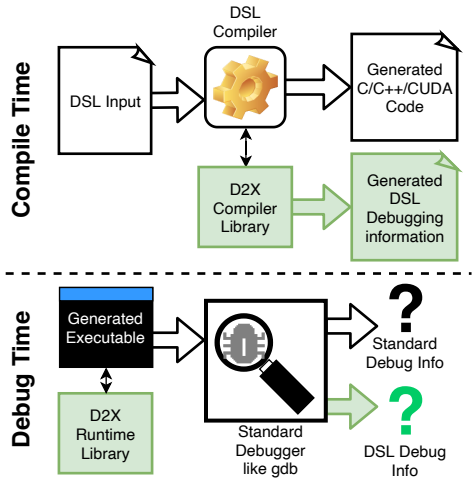


Figure 7.3: The overall system overview for the D2X compiler and runtime extensions. Additional components added/generated by D2X are in green.

expert. ii) The modifications required to be made to the debugger are neither easy nor portable. The source code for the popular LLVM-based debugger LLDB stands at 543K lines of C++ code (not including the test cases or LLVM core). Similarly, the source code for GDB stands at 3M lines of C code. The DSL designers would have to modify these large code bases for each new DSL they build. This approach is used by the designers of cuda-gdb, a debugger for NVIDIA GPUs. But cuda-gdb is maintained by a whole team at NVIDIA, which is not feasible for domain experts wanting to rapidly prototype DSLs.

7.2 D2X System Overview

All the above-mentioned issues guide the design of the techniques that go into D2X. I will present an easy-to-use library that requires no modification to the debuggers, displays rich source and variable information for the DSL input and the generated code, allows managing breakpoints based on DSL source locations, and can be easily extended to display semantic information. In this section, I will describe the overall system design of D2X, its various components, and how both the developers and the end-users interact with it. D2X is a C++ library that comprises two main parts - the D2X compiler library (D2X-C) and the D2X runtime library (D2X-R), and an auxiliary set of helper macros. Figure 7.3 shows the overall working of D2X. Once again, C++ is the ideal choice for D2X since, as we have seen before, it is the language of choice for high-performance DSLs and runtimes. Implementing this library in C++ also allows for easier integration into BuildIt, opening up the world of rich debugging capability to many more DSLs. I will explain each of the components below -

7.2.1 D2X Compiler Library

The D2X compiler library (D2X-C) is the part of D2X that is used by the DSL compiler designers to encode domain-specific information in the generated code. The library API contains functions to encode source information and variable information for each line of generated C or C++ code. The

library then organizes and dumps this information alongside the generated code as C++ arrays and objects. Figure 7.3 shows the role of D2X-C in the code generation process. The developer has to make minor modifications to the DSL compiler code base to call D2X-C In the later sections, we will see that the amount of modifications required for the DSLs I modified is minimal.

7.2.2 D2X Runtime Library

The D2X runtime library (D2X-R) is the part of D2X that is included and linked with the generated code. The library interprets the debug information generated by D2X-C and works with the debugger to provide DSL debugging features to the end-user. The D2X-R library API contains various functions that the user can call from the debugger to obtain source and variable information and insert and delete breakpoints in the DSL. Besides the requirement to compile the generated code with regular debug information (using `-g` for GCC and clang), D2X-R does not add any runtime overhead. This makes D2X very practical for debugging high-performance DSLs, both for correctness and performance.

7.2.3 Debugger Helper Macros

D2X also includes a small set of helper macros to be used with the popular debuggers GDB and LLDB. These are optional and allow the end-user to invoke D2X functionality without having to type long commands. These macros are independent of the DSL and need to be written once to support all the DSLs being debugged using that debugger.

7.3 D2X Implementation

I start by making an observation that most high-performance DSL developers and end-users are familiar with using low-level debugging tools like GDB, LLDB, and WinDbg. In the previous sections, I presented case studies of DSLs built by several collaborators and domain experts. All of them have some familiarity with existing general-purpose debugging tools, but find themselves not sufficiently capable of modifying them or writing their own debuggers. Thus, instead of making a new ad-hoc debugger and debugging formats, D2X actually works directly with unmodified off-the-shelf debuggers.

Before I explain how D2X works, let us understand how these typical general-purpose debuggers work and then try to extend the workflow. Most low-level debuggers take as input the runtime state of the program (registers, memory, instructions) and map them to the source code in the input language. These debuggers use the debug information produced by the compiler alongside the generated code to perform this mapping. D2X takes this idea further and performs a secondary mapping from the source line in the generated code to the DSL context, which includes the DSL input locations, DSL compiler internal state, and the runtime values represented in a way that makes sense for the DSL. To perform this mapping at debugging time, D2X-R uses debugging data tables that are in turn generated by D2X-C alongside the source. Figure 7.4 shows this two-stage mapping. Because this mapping takes the source location as input, D2X doesn't have to deal with low-level instructions and registers or extend complex debug information formats. I will now explain how each of the components of D2X works together.

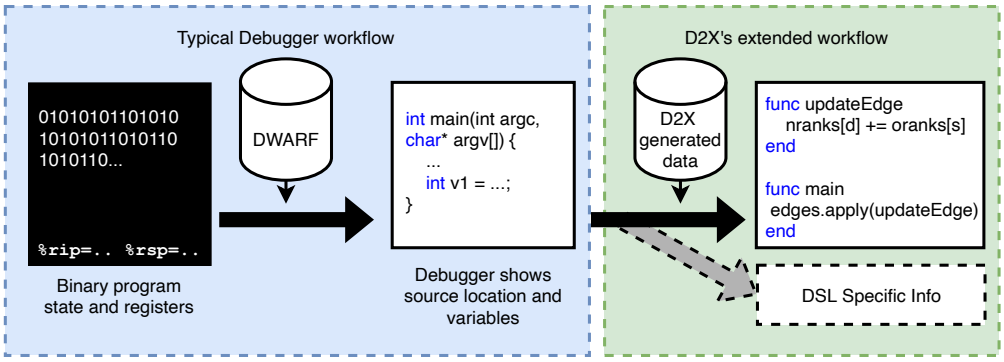


Figure 7.4: Two-stage mapping of source information enabled with D2X. Left (blue) shows typical mapping from binary state to source using DWARF. Right (green) shows mapping from generated source to DSL input using data generated by D2X-C.

7.3.1 Implementation of D2X-C

As explained above, D2X-C is available as a C++ library that can be directly used inside traditional DSL compilers. The developer calls the D2X-C API to encode DSL-related debug information. The complete API is listed in Table 7.1. Since the D2X mapping to debug information is done on the source location of the generated code, D2X-C generates debug info for each line of generated code. Fundamentally, two key pieces of information need to be encoded: i) the source location in the input DSL, and ii) Live variables and their values.

To begin generating D2X debug data tables, the developer instantiates a **`d2x_context`** object and calls **`begin_section()`**. Now, for each new line of code that the DSL compiler generates, they call the **`nextl()`** member function. All calls to other functions between two subsequent calls to **`nextl()`** encode debug information for a single line of source generated. As a result, the developer has to be very careful when emitting newlines in the generated code so as not to misalign the generated code and the debug information.

We discussed above with the example of the program from GraphIt, how the source location in the DSL can be more elaborate than a single line of source code. As a result, D2X is designed to allow encoding a set of line numbers and filenames for each generated line of code. This is achieved with the help of the **`push_source_loc`** function, which can be called multiple times per generated line of code. When queried in the debugger, these source locations are presented as a stack, which the end user can move up and down, akin to GDB frames.

After source locations, the next important piece of information to encode is variables and variable states, which depend not only on the compiler state but also need to access the runtime state. Variables in D2X follow a basic key-value model. Each DSL variable is identified by a key, which is a string. The key of the variable doesn't need to correspond to an actual variable in the DSL source and can thus be used to encode arbitrary information. The values for D2X variables can be of two types. The first type of value is constant strings. These can be used to encode the compiler's internal state that does not change during the execution. An example of such a value is the result of the data flow analysis in the compiler, like we saw in Chapter 6. The second type of value can be a runtime value handler **`rtv_handler`**, which is a lambda that is evaluated at runtime to obtain values of a variable. Each key can be associated with a different lambda that gets the name of the variable as a

Function name	Description
`d2x_context::d2x_context(void)`	Constructor for the D2X context object
`d2x_context::begin_section(void)`	Function to start a new section. All the newlines inside the section need to be tracked by calls to nextl(). Lines outside sections need not be tracked
`d2x_context::end_section(void)`	Ends a previously started section
`d2x_context::nextl(void)`	Tells the context that a newline has been inserted in the generated code. Live variables are automatically inserted
`d2x_context::push_source_loc(string filename, int line_number, [string function])`	Push a source location of the source stack. Can be called multiple times per generated source line
`d2x_context::set_var(string key, string value)`	Inserts a new variable at the current line with a constant string value
`rtv_handler::rtv_handler( function<rt::string(rt::string)> handler_lambda)`	Constructor for a new runtime value handler. Can be used for multiple key-value pairs. The lambda takes the key of the variable and returns a string
`d2x_context::set_var(string key, rtv_handler value)`	Inserts a new variable at the current line with a runtime value
`d2x_context::create_var(string key)`	Creates a new variable in the current scope
`d2x_context::push_scope(void)`	Pushes a new live variable scope
`d2x_context::pop_scope(void)`	Pops a scope and deletes all the live variables in the current scope
`d2x_context::update_var(string key, rtv_handler value)`	Updates the value of a currently live variable
`d2x_context::update_var(string key, string value)`	Updates the value of a currently live variable
`d2x_context::emit_section_info(ostream &oss)`	Outputs the debug information table to the given output stream for the last section
`self_source_loc(void* ptr)`	Obtain the source location for a given instruction pointer in the current program

Table 7.1: Table showing the API functions from the D2X compiler API (D2X-C). Argument names shown in [] are optional. The `rt::string` type is just BuildIt's dynamic type wrapped around `std::string` (`dyn_var<std::string>`).

string and produces the value as a string. To simplify writing and generating these lambdas, D2X simply asks the developer to use BuildIt to implement and generate these lambdas. Table 7.1 shows the `set_var()` member function that can be used to create key-value pairs at any generated line. Since variables are typically live for multiple lines, D2X-C also allows the creation of live variables that are automatically inserted at every line till they are deleted. These variables can also be scoped to mimic the scopes in the DSL and the generated code. The member functions `create_var()`, `delete_var()`, `push_scope()`, `pop_scope()` and `update_var()` help with simplifying variable creation and update.
The `rtv_handler` also has access to a runtime API that lets the handler obtain addresses of variables on the stack given its name by decoding the DWARF information. The `rtv_handler` is one of the key mechanisms for extensibility in D2X. Since the key-value model is very flexible and the developers can supply arbitrary code in the handlers, developers can use this API to implement custom commands that can run at debug time and produce output. These commands can also be used to update the state of the variables if required, like in traditional general-purpose debuggers. In the later sections, we will see how the `rtv_handler` displays sparse data structures stored in multiple formats.
Once all the debug information is encoded, the DSL compiler calls the member functions `end_section()` and `emit _section_info()`, which convert this information into C++ arrays and structures and generate them into the C++ file specified. The D2X-R functions access this source and variable information by reading the generated arrays. I will explain this in the next section.

Finally, the D2X-C library also provides a utility function `self_source_loc` that identifies the source location for the program itself using D2X-C, given a code pointer. This utility is useful for obtaining source locations in DSLs that are written in an embedded way, like in BuildIt. However, since D2X has already been integrated with BuildIt, none of the above steps are required, and all DSLs built with BuildIt get debugging support for free.

7.3.2 Implementation of D2X-R

As explained above, D2X is designed to work with commonly used debuggers because both DSL developers and end-users who have experience writing high-performance code by hand are familiar with their interface and commands. One of the main challenges when dealing with popular debuggers is that their implementation is huge and complex, and modifying them to support the DSLs is a non-trivial effort. Even for debuggers that allow implementing plugins, this approach is neither portable nor scalable. Moreover, it is imperative that D2X enables the DSL designers to add more features to the implementation as they wish. These constraints motivate a novel design for the D2X debugger runtime.

During the design and development of D2X, specifically D2X-R, I inspected all the popular modern debuggers (GDB, LLDB, and WinDbg) and made an observation that all these debuggers implement a feature where arbitrary functions from the program being debugged can be invoked from the debugger when the execution is paused at a breakpoint or a fault. For example, GDB implements the `call` command for this functionality. The invoked functions can run arbitrary code that reads and allocates memory, prints output, or even performs file I/O. D2X exploits this feature of the debuggers to add the features and commands it needs at runtime. D2X injects its own functions into the executable of the program by requiring the end-user to link against D2X-R. Remember, this is the executable of the program written by the end-user, not the binary of the DSL compiler itself. These functions that D2X injects have a well-defined interface and allow the program itself to

present a debugging view of the state of the program. The debuggers also allow the command line to directly read registers like the instruction pointer and the stack pointer as meta-variables and pass them to these functions. All the commands that D2X provides are implemented as just calls to these functions. These functions use the passed instruction pointer and stack pointer to identify where the execution currently is, using the existing debug information, and then use the data generated by the D2X-C to map it to the contextual debugging information encoded by the DSL designer. Figure 7.5 shows an example of one such command - **`xbt`** and how it is invoked. Note that the **`rip`** and the **`rsp`** supplied are from the current stack frame. This means that the end-user can also navigate the stack frame up and down using the usual debugger commands and orthogonally call these functions on each frame. The figure also shows, in the last line, a simplified way of calling the long command using the macros defined (**`xbt`**)

```
namespace d2x_runtime {
    void command_xbt(void* rip, void* rsp);
}
// GDB debugger command interface
(gdb) call d2x_runtime::command_xbt(rip,rsp)
(gdb) xbt
```

Figure 7.5: The definition of a D2X-R function **`d2x_runtime::command_xbt`** and macro with how it is invoked from the debugger at a breakpoint.

The main benefit of this approach is that the entire implementation of the debugger extension is simply written as C++ code that is linked into the executable, thus requiring no modification or plugins to the debugger itself. This also makes D2X's approach very portable and easy to extend as long as the debugger supports calling functions from the executable. Unlike some commonly used ways to achieve the same effect by inserting preprocessor source line annotations in the generated C and C++ code, which only allows for a one-to-one mapping between the DSL input and the generated code, D2X's approach allows the DSL developer to encode arbitrary data and process it in any way they like in the debugger. Most importantly, D2X also works with multi-threaded programs, which is typical for most high-performance DSLs without any modifications as long as the debugger has the ability to pause individual threads. Table 7.2 shows all the commands that can be invoked from the D2X-R API and their description. Next, I will explain these commands and how they simplify debugging in detail.

Displaying extended stack

One of the main features of D2X is to show the "extended stack," which roughly corresponds to the sequence of calls in the DSL that led to the generated source line. Recall that the D2X-C allows encoding this information using the **`push_source_loc`** member function. The extended stack is associated with each source line that is generated. This means that when the execution is paused inside a debugger, each execution frame has a different complete "extended stack" associated with it. The **`xbt`** command shows the extended stack for the currently selected frame. Besides viewing the extended stack, the **`xlist`** command shows the actual source code for the extended stack location. The **`xlist`** shows the source for the current extended stack frame, which can be viewed and changed

Command Macros	Description
`xbt`	Displays the extended stack associated with the current execution stack frame. It can be called at any frame to display the extended stack at that frame.
`xframe [xframe_id]`	Displays the currently selected frame in the extended stack associated with the current execution stack frame. Optional parameter `xframe_id` changes the selected extended stack frame before displaying
`xlist`	Displays the source (DSL input) for the top frame in the extended stack associated with the current execution stack frame. Can be used in combination with `xframe` to inspect the source for all the extended frames.
`xbreak`	Insert a new break point at the specified source location in the DSL. Lists all current breakpoints if called without arguments.
`xdel`	Delete a breakpoint in the source DSL identified by the breakpoint ID.
`xvars [var_name]`	Displays all the contextual variables at the current frame. Value for `var_name` variable is evaluated and displayed.

Table 7.2: Command macros that can be invoked from the debugger and their descriptions. The optional arguments are shown in `[]`. Each of these macros invokes the corresponding D2X-R API function with the `rip` and `rsp` as parameters.

with the `xframe` command, similar to the typical GDB command `frame`. D2X's design decision of allowing the DSL designers to associate an entire stack with each source location allows displaying rich DSL contexts like calling context in UDF for GraphIt and the entire static stage stack in BuildIt. This stack can also be used to mix and show the scheduling information for the line of code stored in a different file.

Displaying extended variables

As explained before, the variables in D2X are key-value pairs that can be evaluated and printed in the debugger. Just like the source location, each generated line of code has a set of variables associated with it. The names of the variables associated with the current execution frame can be printed using the `xvars` command. Supplying the name (key) of a variable also evaluates and prints it. If the variable is a constant stored as a string, it is simply printed out. If the value is an `rtv_handler`, the handler is evaluated and the output is printed.

Inserting and deleting breakpoints

Managing breakpoints based on the source locations in the input DSL is critical to allow the end-user to debug their programs with the least effort. Since one line of input DSL code often corresponds to tens of lines of generated code, it is incredibly difficult to manually insert and remove breakpoints in the generated code. The technique of invoking D2X-R functions from the debugger that print information is not enough to manage breakpoints. D2X needs the ability to invoke debugger commands as a result of the D2X-R API call. To enable this, D2X leverages the `eval` command available in most debuggers. The `eval` command accepts a `printf` style format string with parameters and runs it as a command. The parameters can also be the result of a function from the program

being debugged. Instead of invoking the functions from D2X-R using the `call` command, D2X now invokes it as `eval "%s", d2x_runtime::command_xbreak(rip, ...)`. The `command_xbreak` function takes as parameters a source location in the DSL and looks up the locations of all corresponding generated statements. It then creates a string containing commands to insert breakpoints at those locations. This string is returned and is evaluated as a debugger command. This way, D2X-R is not only able to print relevant information, but it is also able to take control of the debugger and create breakpoints. Breakpoints are deleted in a similar way using the `xdel` command.

The commands introduced in D2X are similar to the commands in typical debuggers to make it easy for end-users to adopt them. The commands listed above can be orthogonally used with existing debugger commands.

7.3.3 DSL Specific Extensions to D2X

Besides being easy to use for both DSL designers and end-users, another key design principle for D2X is to allow DSL-specific extensions. Simple extensions can easily be implemented using the `rtv_handlers` that can be used to run arbitrary code at debug time. The DSL designers can create DSL-specific commands by adding special variables that can be evaluated using the `xvars` command. But besides the `rtv_handlers`, the data generated by the D2X-C library and the D2X-R functions are regular C++ data and code. This means that the DSL developer can extend the debugger further by generating more data from the compiler and implementing functions to decode and display it. This would be difficult to do with existing debugging formats like DWARF and would require messing with the large code bases of the debuggers, hurting the productivity of the DSL designers.

7.4 D2X Case Studies

So far, we have discussed the system design, the various components of D2X, and how they are implemented using D2X's novel design that doesn't require any changes to the runtime. Next, I will walk through two case studies. The first one of the GraphIt DSL standalone compiler to add graph domain-specific debugging information, including displaying source and context information and displaying complex data structures like `VertexSet` that can be implemented as a combination of different data structures in the generated code. Second, I will show D2X can be applied to the BuildIt multi-staging framework itself to display the source and the `static_var<T>` variable state from the *static* stage. This allows the users of BuildIt to perform end-to-end debugging while writing code in multiple stages, starting right from the source of the *static* stage to the execution and output of the *dynamic* stage. Not only this, DSLs built with BuildIt, like shown in Chapter 6, would not automatically get debugging support without having to change a single line of code in the implementation. For both the case studies, I will also quantify the amount of work required to apply D2X in terms of the lines of code changed to demonstrate that D2X is an easy-to-adopt system.

7.4.1 Applying D2X to the GraphIt Compiler

GraphIt is a DSL for graph computations that generates high-performance C++ and CUDA to be run on CPUs and GPUs, among other hardware. GraphIt separates what is computed (specified in

the algorithm language) from how it is computed (specified in a scheduling language). The GraphIt algorithm language has high-level operators such as the `edgeset.apply` and `vertexset.apply` that are lowered to low-level code after applying a series of transformations and analyses to better suit the implementation of the operators for the overall application and the graph inputs. GraphIt is a classic example of a DSL where a single line of input code maps to multiple lines of complex low-level code spread out in multiple places in the generated code. The generated functions are also specialized for each call site differently. All these factors make it incredibly hard to debug the generated code by simply attaching a debugger to the generated code.

The DSL also uses high-level sparse data-structures like the `edgeset`, `vertexset` and `PriorityQueue` that can be lowered to one or more different representations. Debugging the state of these data structures at runtime is not as straightforward as printing a variable in the debugger. I will explain below the two main capabilities we add to GraphIt using D2X and how they appear in the debugger and improve developer productivity.

Source locations in GraphIt

Since this implementation of GraphIt, unlike the one described in Chapter 6, is a standalone DSL, developers using GraphIt write the algorithm input in a `.gt` file that is parsed by the GraphIt frontend. The frontend already records the line and column number for each operator it parses for printing error messages. This makes it easy to support D2X extensions without modifying the parser. The changes made to the GraphIt compiler now propagate the line numbers from the parser through all the mid-end passes to the code generation phase, which is modified to insert calls to the D2X-C API and insert source locations for each generated C++ line. To provide more context for the specialization of User Defined Functions (UDFs), the compiler also inserts the line number of the call site for which a particular UDF is specialized. An example GraphIt source for the PagerankDelta application, generated code, and the information displayed inside GDB is shown in Figure 7.6. Notice for a source location inside the `updateEdge` UDF, the extended call stack shows the location of the operator for which this UDF is specialized.

Debugging complex data structures

One of the key data structures in the GraphIt DSL is the `VertexSet` that holds the active set of vertices to be processed in each round. The performance of the entire algorithm depends a lot on the implementation of this data structure in the generated code. As a result, GraphIt uses 3 different representations - Bitmap, Boolmap, and CompressedQueue, each offering different tradeoffs in terms of performance. Different representations are suitable for different parts of the applications. Sometimes, as the number of actually stored vertices increases, GraphIt switches representations for this data structure. Debugging this data structure thus requires checking the current representation and decoding it. D2X supports debugging `VertexSets` with the help of a `rtv_handler`. Figure 7.7 shows the handler declared for debugging the `vertexset` data structure. This handler finds the data structure on the stack using the name of the variable (Line 2), then checks the current format it is stored in (Line 6) and finally serializes the elements stored to produce an output (Line 8/Line 12) (Boolmap and Bitmap are always updated together, so a single check is enough). Figure 7.6 shows the produced output. We can see the contrast between the output produced using the `rtv_handler` and the usual `print` command available in the debugger. The usual `print` command just shows the

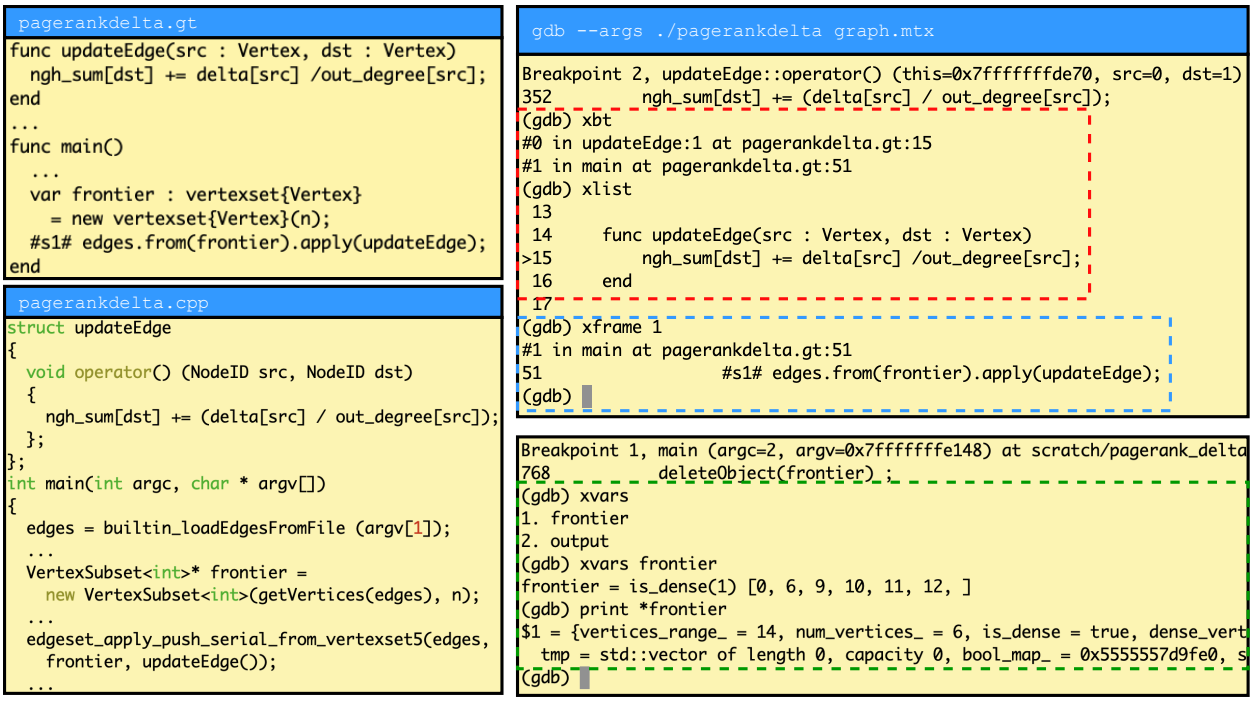


Figure 7.6: The GraphIt DSL input and the generated code for PagerankDelta and the view from the debugger (GDB). The red box shows the extended stack and the listing using **`xbt`** and **`xlist`** commands. The blue box shows the UDF calling context using **`xframe`**. The green box shows the **`vertexset`** objects and the output from the **`rtv_handler`** using the **`xvars`** command.

```
d2x::rtv_handler frontier_resolver([&] (auto v) -> auto {
  dyn_var<frontier_t**> addr = find_stack_var(v);
  dyn_var<frontier_t*> set = addr[0];
  dyn_var<string> ret_val = "is_dense(" +
    to_str(set->is_dense) + ") [";
  if (set->is_dense)
    for (dyn_var<int> i=0; i < set->num_vertices; i++) {
      ret_val += to_str(set->dense_vertex_set[i]) + ",";
    }
  else
    for (dyn_var<int> i=0; i < set->vertices_range; i++) {
        if (set->bool_map[i]) ret_val += to_str(i) + ",";
    }
  return ret_val + "]";
});
```

Figure 7.7: Definition of the **`rtv_handler`** to display a **`vertexset`**. Notice the check on the format and the serialization based on it.

Component	Lines of C++ Code
GraphIt DSL Compiler and Runtime	45,966
Delta for adding D2X support	667 (+597/-70)
GraphIt percentage change	**1.4%**
D2X-C	465
D2X-R	956
D2X helper macros	40
D2X total	**1452**

Table 7.3: The number of lines of code changed in GraphIt and the number of lines of code required for the implementation of D2X. The support for contextual debugging to GraphIt was added by changing merely 1.4% of the lines of code.

struct and its members and leaves deciphering them completely up to the end user. D2X's approach for debugging complex data structures makes it incredibly easy for the end user to stop the execution at any step and check the state of the algorithm.

Furthermore, Table 7.3 shows the number of lines of code changed to support D2X debugging in the GraphIt DSL compiler. This includes all the lines changed for propagating the debugging information through the compiler passes and the calls to D2X-C to generate the debug information. We can see that the changes are only 1.4% of the entire GraphIt code base, supporting the claim that D2X boosts end-user and DSL developer productivity with very little effort.

7.4.2 Applying D2X to the BuildIt Framework

In the next case study, the BuildIt multi-staging framework itself is enhanced to support recording the debugging information from the *static* stage and expose it as extended debugging state in the *dynamic* stage using D2X. This effort makes it so that all DSLs written with BuildIt as described in Chapter 6 get support for end-to-end debugging automatically without any changes. This effort makes the BuildIt system itself more useful and easier to adopt. Just like the previous GraphIt DSL, each generated line of code in the *dynamic* stage can be a result of an entire call stack in the *static* stage as opposed to a single line. Furthermore, the `static_var<T>` variables are completely erased from the generated code, and are not visible when the user tries to debug the generated code. However, these variables definitely have an effect on the generated code since they can be used to specialize the code generated. Providing access to the state of these variables at each location would be essential to the complete debugging experience.

First Stage Source Access

As we have thoroughly described in Chapter 3, BuildIt's internal logic uses a very simple idea to implement staging. Each expression and statement that is extracted is assigned a "static tag" that uniquely identifies the statements from other statements in the generated code. This mechanism of static tags is crucial to code deduplication, branch collapsing, and detecting loops in BuildIt. The static tag is comprised of two separate parts. The first part is the call stack in the first stage code at the site where the expressions were created, and the second part is the state of all static variables

```
// First stage BuildIt code
dyn_var<int> power_f(dyn_var<int> base, static_var<int> exponent) {
  dyn_var<int> res = 1, x = base;
  while (exponent > 0) {
    if (exponent % 2 == 1)
      res = res * x;
    x = x * x;
    exponent = exponent / 2;
  }
  return res;
}
// Generated code with exponent = 15
int power_15 (int arg0) {
  int res_1 = 1;
  int x_2 = arg0;
  res_1 = res_1 * x_2;
  x_2 = x_2 * x_2;
  ...
  x_2 = x_2 * x_2;
  return res_1;
}
```

Figure 7.8: The first stage BuildIt source code for the power function implemented with repeated squaring and the generated code with **exponent** specified as 15. The **static_var<int>** exponent is completely erased from the generated code but produces a sequence of statements.

that were live at the site where the expression was created. The first part packs all the information we need to generate D2X source information. D2X-C's helpful **self_source_loc** util API is used to obtain the source location for each static tag and encode it for each line of generated C++ code using the D2X-C API. This small change is enough to encode the entire source information from the first stage.

First Stage Variable Access

Encoding the state of the first stage or the **static_var<T>** variable follows a similar procedure. Each statement in the generated code is already attached with a snapshot of all the live **static_var<T>** variables. D2X simply serializes each **static_var<T>** variable from the snapshot into a string and encodes it as an **xvar** using the D2X-C API.

Figure 7.8 shows the input and generated code for a power function using repeated squaring. Figure 7.9 shows the output of various usual debugger commands and the D2X commands. The **bt**, **frame**, and **print** commands show the second stage source and variables, while the **xbt**, **xlist**, and **xvars** commands show the first stage source location and variables. For example, at different lines in the generated code, the **static_var<int>** variable **exponent** has different values attached to it, as it would have in the first stage code. Finally, the **xbreak** command shows how inserting one breakpoint in the first stage code inserts 3 breakpoints at the corresponding generated lines of code.

```
(gdb) bt
#0  power_15 (arg0=3) at power_test.cpp:11
#1  0x0000555555556c52 in main (argc=1, argv=...) at ...
(gdb) frame
#0  power_15 (arg0=3) at scratch/power_test.cpp:11
11        x_2 = x_2 * x_2;
(gdb) xbt
#0 in power_f at power.cpp:16
#1 in _M_invoke at std_function.h:316
(gdb) xlist
 14                    if (exponent % 2 == 1)
 15                            res = res * x;
>16                    x = x * x;
 17                    exponent = exponent / 2;
 18             }
(gdb) xvars
1. exponent
(gdb) xvars exponent
exponent = 7
(gdb) print res_1
$1 = 27
(gdb) xbreak 15
Inserting 3 breakpoints with ID: #1
Breakpoint 2 at: scratch/power_test.cpp, line 8.
Breakpoint 3 at: scratch/power_test.cpp, line 10.
Breakpoint 4 at: scratch/power_test.cpp, line 12.
```

Figure 7.9: Debugger (GDB) output for the code generated in Figure 7.8. The output of the **bt**, **frame**, and **print** command show the second stage source and variables. The output of the **xbt**, **xlist**, and **xvars** commands shows the first stage source and the state of the **static_var<T>** variables. The **xbreak** command shows breakpoints inserted in the first stage.

```
el::tensor<int> c({M}, output);
el::tensor<int> a({M, N}, matrix);
el::tensor<int> b({N}, input);
b[j] = 1; // Initialization  of b
c[i] = 2 * a[i][j] * b[j]; // Use in matrix x vector
```

Figure 7.10: An input for Einsum lang implemented on top of BuildIt that initializes a vector and performs matrix-vector multiplication. The example demonstrates the use of constant propagation analysis and specialization.

```
BT 1, main (arg0=1, arg1=...) at einsum_test.cpp:25
25       output_4[i_5] = output_4[i_5]
            + ((var20 * matrix_2[(i_5 * 8) + j_6]) * 1);
(gdb) xbt
#0 in create_increment at einsum_matrix_mul.cpp:161
...
#6 in operator= at einsum_matrix_mul.cpp:229
#7 in m_v_mul<16, 8> at einsum_matrix_mul.cpp:370
(gdb) xframe 7
#7 in m_v_mul<16, 8> at einsum_matrix_mul.cpp:370
370             c[i] = 2 * a[i][j] * b[j];
(gdb) xvars
1. b.constant_val
(gdb) xvars b.constant_val
b.constant_val = 1
```

Figure 7.11: The D2X output from the debugger for the program in Figure 7.10. The **`xbt`** command shows the steps inside the DSL implementation, and the **`xvars`** command shows the details of the analysis stored as **`static_var<T>`**.

Debugging DSLs with BuildIt

Next, let us take the example of a simple Einsum expression DSL compiler written on top of BuildIt, using the procedure described in Chapter 6. This DSL generates code for expressions on tensors written with einsum notation (like **`a[i][j] = b[i] * c[j]`**) and is a mere 330 lines of code. The entire implementation of this DSL is available on the BuildIt website [27]. The implementation also performs constant propagation in tensors by the use of static variables using the methodology described in Chapter 5. D2X uses the generated debugging information to inspect this internal state of the data-flow analysis. The embedded DSL is tested on an input program (Figure 7.10) where a rank 1 tensor **`b[i]`** is initialized to all 1s. This tensor is then used in another computation that performs matrix-vector multiplication. The DSL compiler performs constant propagation through the **`static_var<T>`** variables. Figure 7.11 shows the extended stack and frames that show how each step of the generated code is produced. Using **`xvars`**, we can also see the propagated constant value inside the debugger. Note that not a single line of change was made to the DSL implementation apart from the above modifications made to BuildIt. This further supports our claim that D2X improves DSL designer and end-user productivity. Table 7.4 shows the total number of lines and the lines changed.

Component	**Lines of C++ Code**
BuildIt DSL compiler framework	11,205
Delta for adding D2X support	428 (+407/-21)
BuildIt percentage change	**3.81%**

Table 7.4: The number of lines of code changed in the BuildIt code base to support D2X debugging information generation. Notice that besides these changes, no other changes are required to the DSLs built on top of BuildIt.

This concludes our discussion of the D2X debugging system and its applications to standalone DSL compilers and compilers written with the BuildIt framework. I conclude that implementing debugging support for DSLs is critical to improving productivity and can be achieved by very simple modifications using the D2X system. In the future, any DSL written with BuildIt comes with debugging support built in for free.

Chapter 8

Related Works

In this thesis, I have covered many different but related topics, including high-performance abstractions like libraries and compilers, domain-specific languages (DSLs), or languages that are focused on facilitating writing applications from a specific area, multi-staging, and their different categories. I compared different existing implementations of multi-staging and the benefits and drawbacks they offer. I covered different components of writing a compiler, including analysis, transformations, and hardware-specific code-generation. I then covered the design and construction of DSLs from three very different domains: graph analytics and regular expressions. Finally, I introduced techniques to add debugging support to DSLs to improve end-to-end user experience.

Many of these topics have been the focus of research for years. Multi-Staging, the core idea on which this thesis hinges, has been around for more than 3 decades. In this Chapter, I will present a discussion of these related works and how my research fits into the bigger picture. I will break down the related works into sections broadly - Multi-Staging, Domain Specific Languages, Compilers and Code Generations, Graph Analytics Libraries and Compilers, Network Protocol Optimizations, Regular Expression Libraries and DSLs, Debugging Techniques and Infrastructure.

8.1 Multi-Stage Programming

One of the earlier and most comprehensive references to multi-staging comes from Walid Taha's tutorial on A Gentle Introduction to Multi-Stage Programming[168, 169]. This work introduces many of the ideas used in this thesis, including multi-stage programming and implementing compilers and DSLs using stage interpreters with Futamura projections [69]. This work is heavily based on and inspired by the BUILDER library [162] for the SUIF [193] compiler system, which, as far as I 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.

In Chapter 2 under Figure 2.1, I have provided a list of the most popular frameworks that provide or use multi-staging. I have categorized them based on the exact flavor of multi-staging they implement. A lot of the design decisions for BuildIt have been based on addressing the shortcomings of these frameworks. As shown in the BuildIt row of the table, BuildIt is designed to be extremely seamless while addressing the side-effect leakage problem present in most imperative staging frameworks.

Lightweight Modular Staging (LMS) [151] is the closest work to BuildIt and creates a staging system in Scala. It is also applied for code generation and embedding DSLs. Since the LMS system solves the control-flow problem by taking a hybrid (compiler + execution) approach, and due to its type-based interface, LMS also comes very close to being one of the most seamless frameworks. However, since it is embedded in Scala, its applicability to high-performance domains remains limited due to the fact that high-performance domains requiring low-level optimizations are more suitable for languages like C++. A lot of existing code bases for such high-performance libraries are also in C++. Just like BuildIt's data-flow analysis, LMS has also been applied for achieving fast, modular whole program analysis using staged abstract interpreters [190]. Although the techniques aren't exactly the same, the benefits for DSL design are similar. LMS has also been shown to be applicable in building verified staged interpreters and thus verified compilers [18]. This is a direction I, too, would like to pursue for BuildIt in the future. Such a work could be key to unlocking the tri-fecta of verified, fast, and easy-to-implement compiler abstractions.

Another major application of multi-staging and extracting programs using executions is in the area of machine learning for describing and generating code for neural network models. The two popular frameworks in this area, Pytorch [138] and Tensorflow [175], are both embedded in Python. These frameworks have gone through several changes in methodologies, from using purely overloaded operators to extract static graphs to extracting Python's AST representation to extract control-flow and dynamism. Recently, Pytorch 2.0 has also used the technique of hijacking the Pytorch interpreter (Python's equivalent of the evaluation procedure) to aid with the extraction process. These languages then compile these extracted graphs down to highly optimized hand-written machine learning kernels, improving overall performance. The extraction process also aids with auto-differentiation since it allows for automatically generating the backward passes, improving the user experience greatly.

Apart from the above embedded frameworks, a lot of work has been done on building languages with multi-staging as a first-class feature. Scheme [163] probably is the earliest known language in this area that takes the approach of treating data and program as the same representation, providing the basic primitives to build multi-staging. Specialized multi-stage languages like MetaML [170], MetaOCAML [29], and Mint [191] 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 means of annotations or specialized syntax. MetaML and MetaOCAML either have a lot of code duplication [30] [37] [166] due to continuous style monadic execution or have to handle side effects through the means of a global state or delimited control operators. Even so, some of these suffer from the side-effect leak problem or simply do not allow side-effects on *static* state under control, flowed based on *dynamic* variables, which we have seen is critical for many applications like interpreters. Terra [56], [57] is a meta-programming language that leverages a popular scripting language, Lua, to stage its execution. It also uses a compiler-based extension to add the staging-related features. The compiler-based approach and a choice of host language not suitable for high-performance programming remain a hurdle for adoption in DSL development.

Under C++, templates and consteval [180] and concepts are the language-based solutions to support meta-programming. Templates, although Turing complete, are often considered clunky due to the syntax and extremely difficult to debug due to the only view into the first stage being the generated cryptic C++ error messages. Consteval was added to solve these problems and greatly improves the syntax, bringing the syntax from the two stages closer together, improving seamlessness. However, even though the syntax gap has been bridged, semantically the two

remain pretty distant with heavy restrictions on features available under consteval execution . The taskgraph-metaprogramming library [11] is another attempt at bringing multi-staging into C++. Similar to SUIF, it uses operator overloading for expressions and specialized macros for control flow. As expected, it suffers from the side-effect leak problem, but is valuable in cases where the control dependence from dynamic to static state doesn't exist. Most importantly, taskgraph provides an accompanying scheduling language that allows for optimizations post-extraction, making it a good fit for high-performance domains. Intel's ArBB [132] enables runtime generation of vector-style code using a combination of operator overloading and macros in C++.

Finally, the simplest form of multi-staging, text-based multi-staging is used everywhere - Web server languages like PHP, NGINX, NodeJS, and ASP.NET use text-based, multi-stage programming for generating client code. While being extremely easy to implement, these techniques usually aren't very seamless. However, more importantly, since the *dynamic* stage program is handled as strings, any type checking of this code has to be deferred to the point when the *dynamic* stage code is compiled and/or executed.

8.2 Domain-Specific Languages, Compilers and Code Generation

High-performance DSLs have played a critical role in optimizing applications from several domains. Halide [147], a DSL for image processing applications, was the first to introduce the idea of separating algorithms from schedules. Other DSLs like GraphIt [202, 199, 26, 22], TACO [107], Tiramisu [6], Taichi [89], Diderot [36] have applied the same ideas to other high-performance application domains. Machine Learning is a domain where a lot of DSLs have been developed to address the ever-increasing need for performance. Pytorch [138], Tensorflow [175], TVM [35] generate low-level optimized code that targets CPUs, GPUs, and distributed settings. These frameworks offer optimizations like fusion, reordering, and recompute to obtain the best performance in an automated way without any user intervention or scheduling.

DSLs also find applications in non-performance critical domains where the goal is to provide a higher-level abstraction that is either more ergonomic to use or saves a lot of effort. DSLs for visualization like Tikz [171], Vega/Vega-Lite [43, 44], gnuplot [192], matplotlib [91], Graphviz [149] are popular DSLs from the visualization domain.

Build system and build system generator specification DSLs like make [65], cmake [93], Ninja [123] use a declarative syntax to specify dependencies between source files and targets, along with the commands to produce the targets. Even though high performance is not the primary goal for these DSLs, efforts have been put into making these systems scale for larger projects using parallelization, caching, and even better dependency resolution algorithms. DSLs like lex/flex [140], yacc/bison [98], ANTLR [137] solve a meta problem of using a declarative syntax to specify the grammar for languages and generate efficient parser code for other compilers and DSLs.

Many DSL compilers have been implemented as standalone compilers implementing their own parsers, IR definitions, analysis, and code generation for different architectures. This is the approach taken by DSLs like GraphIt. Halide takes the approach of embedding DSLs in a host language but still implements its own infrastructure for analysis, transformation, and code generation. Specialized frameworks for designing compilers like AnyDSL [42], Delite [32], MLIR [115], and

LMS [151] have been developed for rapidly prototyping DSLs. Some programming languages like MetaOCaml [29], Scala, Scheme, and Lisp have built-in support for implementing compilers through stage programming. Frameworks like LLVM [114] provide toolchain support for implementing analysis, transformations, and code generation for general-purpose languages targeting a variety of architectures.

8.3 Graph Analytics Libraries and Compilers

The graph computation domain has had a lot of research in optimizations for different types of applications targeting different architectures like shared-memory [164, 158, 159, 197, 76, 165, 141, 86, 112, 75, 4, 185, 200], GPUs [126, 135, 12, 186, 121, 133, 102, 101, 189, 83, 87, 120, 157, 85, 52, 160, 131, 33, 104, 70, 82], and manycore architectures [34, 117, 141]. GraphIt [201, 25, 198, 23] is a domain-specific language that presents a high-level bulk synchronous model with operators for working with sets of vertices and edges. GraphIt generates state-of-the-art code for CPUs, GPUs, and other specialization architectures built for sparse computations like Swarm [95] and Hammerblade. The Unified Graph Framework UGF [23] framework introduced a unified graph IR and a graph VM to reuse target-independent optimizations while providing utility functions for lowering to different architectures.

8.4 Network Protocol Optimizations

Optimizing networks stack to obtain the best out of the hardware has a long history of research, both in implementing high-performance hand-tuned libraries for TCP/IP/UP [5, 125, 96, 77, 63, 99, 167, 78] and manually creating custom protocols to better fit the applications and environment [122, 92, 152, 113, 167]. Researchers have proposed algorithms for improving existing features like congestion control [110, 130, 73, 62, 8]. However, the handwritten implementations that add or improve a few features are not able to keep up with the rapidly changing end-to-end application needs that require custom tweaking of the full protocol.

DSL compiler techniques have also been applied within the broader network domain for creating custom protocols [118, 15, 90, 14, 187, 71, 108] as well as for optimizing network applications [72]. While these network DSLs allow customizing some features or layers of the protocol, they are not able to perform whole-stack optimization. In particular, P4 [15], the most popular application of compiler techniques to the networking domain, only focuses on packet forwarding and stateless packet processing for programmable switches. Similarly, Rubik [118] proposes a DSL for programming network stacks for middleboxes. However, Rubik only focuses on optimizing bidirectional traffic flows in middleboxes while also requiring intricate knowledge of compilers. ClickNF [71] extends the ideas in Click [108] to host network stacks. Similar to Rubik, ClickNF takes a traditional compiler approach. Consequently, making adoption of new features and libraries challenging, DSL approaches have also been used to solve the layout optimization problem [181, 100].

8.5 Regular Expression Libraries and DSLs

A lot of research has been done in optimizing regular expressions, both using library and compiler techniques. Perl Compatible Regular Expressions (PCRE2) [144] is an updated version of PCRE - a general and widely used regular expression library used for powering Regex matching in a variety of languages like PHP, R, and Delphi.

RE2 [48] is a regex library developed in C++ by Google. It was implemented for production use; therefore, its main goal is to be able to handle regular expressions from different types of users by limiting the amount of memory used, to avoid problems like stack overflowing [49]

Hyperscan [188] is a high-performance regular expression library developed by Intel. It is primarily optimized for use in deep packet inspection (DPI). Unlike other regular expression libraries, it is highly optimized for multi-pattern matching and streaming. Hyperscan performs graph decomposition on the regex to split it into literal strings and finite automata components. For the literal strings, it supports multi-string shift-or matching, which exploits SIMD operations for parallel matching

8.6 Debugging Techniques and Infrastructure

Some research has been focused on building new interfaces for debugging DSLs [16, 195, 61, 184, 17, 194], but these debuggers either resort to an interpreter-based execution while debugging, which affects the performance, or build their debugging infrastructure tied to one particular debugger, lacking portability. Other works have focused on performance [119] but typically have a narrow focus, like concurrency, and do not adapt to the needs of upcoming DSLs. General-purpose language debuggers have been around for a while and rely on the compiler generating standardized debugging formats like DWARF [40] or PDB. These debuggers and the format standards are often very rigid, and extending them is out of scope for typical domain experts. There has been work on verifying and providing formal guarantees about the debuggers [58], but these guarantees are very hard to extend to domain-specific extensions.

Chapter 9

Conclusion and Future Work

In the last couple of years of my research, I have been asked this question multiple times: "Don't we already have a lot of programming languages? Why are you trying to make it easy for people to make more languages?" If we look back at the history of programming languages, the earliest languages were extremely general-purpose. Languages like FORTRAN, COBOL, C, even though they were developed for specific problems, did not exploit domain-specific aspects and were pretty general. This is because the community was still evolving and learning a lot about language design, tradeoffs, and implementation. Developing general languages and solving problems for everyone furthers our understanding of PL. However, in the last two decades, a shift has been made towards DSLs. This is because the community has evolved enough that whatever we have learned from our experience with PL design can now be applied to solve concrete problems.

The natural evolution for any field is when the field gets so mature with ideas that these ideas can be conveyed to the folks on the periphery or even outsiders, and they can start using these techniques. The periphery of PL includes folks who write programs and code but aren't necessarily PL experts. The future I see for my research is this, where PL techniques are being used in small communities to solve high-performance problems within themselves. Instead of creating the next C++ or the next JavaScript or the next Python, I want to empower academic and engineering communities of a handful of experts and 50-100 users to be able to create and share programming abstractions. I don't think these communities are saturated yet and could benefit greatly from such advancements as today they do with high-performance libraries. In fact, we have seen communities see accelerated growth when standardized programming interfaces were created and shared. For example, PyTorch and TensorFlow for machine learning. I believe this is going to be the future of programming language development for the coming decade.

This thesis introduces key ideas to facilitate steps in this direction. I started by uncovering the tradeoffs in the high-performance software development that stems from trying to balance generality, high-performance, and development effort. I proposed multi-staging as a way of getting the best of all the worlds since writing abstractions that generate optimized code using multi-staging require very little boilerplate code, unlike traditional compiler techniques. In Chapter 2, I compared different multi-stage execution frameworks and categorized them based on language choices, ease of translating code, whether they generate source code, what paradigm they support, and what techniques they use for implementation. I identified a key issue, the side-effect leak problem that plagues many imperative multi-stage programming frameworks, and proposed a methodology called Re-Execution Based Multi-Staging (**REMS**) that addresses the issue. In Chapter 3, I introduced BuildIt, an implementation of **REMS** embedded in C++, and described the programming model

and abstraction. In Chapter 4, I described the implementation of BuildIt in a lightweight way using just operator overloading and showed how it satisfies all the requirements of **REMS**. In Chapter 5, I proposed extensions and layers on top of BuildIt that facilitate high-performance development by making it easy to simplify, specialize, analyze, parallelize, stage layouts, and hoist conditionals to generate the most optimized low-level code with minimum effort. In Chapter 6 I described the design and implementation of three DSLs built using BuildIt from a wide variety of domains, including graph analytics, ad-hoc network protocol generation, and Regex matching. All these DSLs showed how the developers can match performance from the state-of-the-art compiler frameworks with 10- 100x less effort. I also discussed some ongoing projects that are using BuildIt to create high-performance abstraction. Finally, in Chapter 7 I introduced D2X, a system that allows adding arbitrary debugging support for standalone DSLs. I also demonstrated applying D2X to BuildIt itself so that any DSLs built with BuildIt would get debugger support for free.

I believe with this thesis, I have created a basic foundation for tools that allow non-compiler experts to write their own programming languages with ease. Furthermore, I have applied these basic tools to a variety of domains to demonstrate the usefulness of the system, and along the way discovered reusable components like a layout customization layer, data-flow analysis using staging, and targeting modern architectures like GPUs and FPGAs. Finally, I have also invested effort in building related tools like D2X and Dynamic-By-Default that further improve the developer experience when working with tools like BuildIt. I believe all these have significantly contributed to the above-mentioned goal of getting PL techniques in the hands of non-compiler experts.
BuildIt is available open source under the MIT License at https://buildit.so

9.1 Scope and Limitations

A large focus of this thesis has been on multi-staging and DSL development. In the Chapters so far, I described the design of three different domain-specific languages implemented using BuildIt that target very different domains. I also discussed some ongoing external works on applying BuildIt to optimizing domains. These DSLs target not only different domains but also a wide variety of different architectures, including CPUs, GPUs, and FPGAs. All these DSLs also show a 10 to 100 times reduction in code size as compared to similar performing compilers implemented using traditional techniques. These case studies talk about the wide range of applicability of the techniques presented in this thesis and the BuildIt system itself. However, there are some limitations which make BuildIt not applicable to certain kinds of DSLs, where either traditional compilers or a different multi-stage system might be more suitable. I will describe broadly the kind of domains and languages that currently cannot benefit from BuildIt's staging capabilities.

9.1.1 BuildIt Offers Limited Syntax

All the DSLs described before, including EasyGraphit, NetBlocks, and the BArray language, are embedded in C++. This is naturally true because BuildIt is a library that uses C++ as its host language. While C++ is a great choice for implementing the optimization for the operators, languages implemented this way are limited by the syntax that C++ offers. C++ has a rich support for building clean abstractions, including function and operator overloading, highly flexible generics based on templates, constexpr, and concepts, and with the recent proposed upcoming support for compile-time

type reflection, which provides plenty of flexibility for most libraries and embedded DSLs. However, C++ has very limited to no support for extending the syntax itself, unlike some other languages like Lisp, Racket, MetaML, Julia, and Rust with its procedural macros.

As a result, if a DSL requires adding lots of special syntax to make the abstraction more ergonomic, BuildIt would not be a good fit. For example a visualization DSL like Tikz that uses a lot of operators which intuitively describe elements of the visualization itself for example, `--` for a straight line, `..` for a dotted line, `<-`, `->`, and `<->` for lines with different arrow positions, `-|` for orthogonal polylines that go horizontally and then vertically and `|-` for lines that go vertically and then horizontally and many more. Now, these operators aren't available in C++ and can be desugared into function calls, but that takes away the intuitive aspect of TikZ that makes it so easy to use. Writing such DSLs by directly embedding them with BuildIt would be extremely difficult, if not impossible.

However, there is a middle ground, where DSL designers can use BuildIt for the optimization and lowering of these DSLs. A custom syntax can be supported in the traditional way by writing a parser and accepting the input program as a string or a file. The parsed IR/AST can then be optimized and lowered using BuildIt's multi-staging capabilities. This is similar to how the BREeze regular expression works, although the regular expression is fairly easy to parse. The MARCH DSL explained above also works in a similar decoupled parsing and lowering implementation, but it uses BuildIt for both the phases. In the future BuildIt can be augmented with a generic parsing library that can streamline end-to-end DSL development. BuildIt can also be implemented in a language like Rust that allows language-level extension of syntax.

9.1.2 Limited Capabilities for Lifting

As seen in all the case studies above, BuildIt is a great and natural fit for languages that are implemented purely with lowering. Lowering refers to the style of compiler design where higher-level abstractions are converted into lower-level abstractions. An example of this is writing a program in a language that supports coarse-grained operations like `map` and `reduce` and lowering them into parallel and serial `for` loops and other operations. The opposite of lowering is lifting, where lower-level code is often pattern-matched into higher-level operations with equivalent semantics. Lifting is often followed by lowering into a different lower-level representation that is more optimized. Polyhedral compilation, where loop nests are lifted into a higher-level abstraction with polyhedra for iteration domains and dependencies, is an example of lifting. Since BuildIt makes the key operations simplification and specialization very easy, lowering is as easy as adding a few functions and their implementations. BuildIt's methodology for fusing operators does allow for a certain level of lifting by using `static_var<T>` variables combined with lazy evaluation to gather the context from nearby operations and convert them into more optimized versions. However, these capabilities are limited.

As a fallback, for DSLs that rely a lot on lifting, BuildIt allows explicit lifting post the staging and lowering by means of a built-in pattern-matching language. Developers can write rewrite rules with expression trees to match and replace parts of the extracted program with an equivalent, more optimized version. However, the support for matching entire statements needs to be added. For more "global" optimizations that require lifting statements from entirely different parts of the program, BuildIt allows developers to write their own AST passes as shown in Chapter 4. While these passes are extremely powerful, they require understanding compiler internals and implementing explicit manipulations, which might not be possible for domain experts.

A somewhat related issue is with BuildIt's approach for data-flow analysis. In Chapter 5 I described a methodology for performing generic data-flow analysis on top of BuildIt by simply tracking abstract facts about *dynamic* values as `static_var<T>` vars. While this works great for forward data-flow analyses, it fundamentally doesn't work for backwards data-flow analysis since values only flow forward in C++. This makes it difficult to implement very critical analyses, like liveness analysis, that can be critical to avoid excessive allocations. I am currently working on an extension to BuildIt with a new type template `nd_var<T>`, which, like `static_var<T>`, would have concrete values in the *static* stage, but the values would flow backwards instead of forward, allowing developers to gather information from statements that haven't yet executed. With this support added, BuildIt users should be able to perform a combination of forward and backwards passes for exploiting maximum optimization opportunities.

9.1.3 Lack of Support for Automatic Optimizations

One of the most important caveats of writing DSLs with BuildIt is that the developer still has to implement all the optimizations themselves. This is usually okay for domain-specific languages requiring domain-specific optimizations. However, there are examples of optimizations that are common to many domains and should be reusable. The Layout Customization Layer explained in Chapter 5 is an example of such a modular optimization. Similarly, the load balancing API built on top of BuildIt's GPU code generation mechanism in the EasyGraphit implementation can find applications across domains with irregular workloads. Work needs to be done in consolidating a "library" of common composable optimizations that domain experts can simply pick and choose, and further reduce the boilerplate for implementing their domain-specific libraries. In some simple cases, new DSLs could be made by simply choosing the optimizations that are valid for the domain.

9.1.4 Narrow Focus on Performance Related DSLs

By design, BuildIt and the techniques presented in this thesis are focused on code generation and optimizations, keeping high-performance as a goal. Even though many DSLs fit in this category, there are DSLs that care about other aspects; some of these DSLs might not even produce executable code. DSLs like build systems (and build system generators), markup languages, visualization and animation DSLs, or even DSLs for managing resources in the cloud fit in this category. BuildIt has very limited support for such DSLs. Even within DSLs that execute but where the focus is on other aspects beyond performance, like verification or a rich type-system, BuildIt's capabilities might be limited. More work needs to be done to explore if any of the techniques developed as part of BuildIt might be applicable to other non-performance-focused domains.

Bibliography

[1] Boost c++ libraries. https://www.boost.org/. Accessed August 18, 2025.

[2] Php manual. https://www.php.net/. Accessed August 18, 2025.

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

[4] Aberger, C. R., Tu, S., Olukotun, K., and Ré, C. EmptyHeaded: A relational engine for graph processing. In *Proc. SIGMOD* (2016).

[5] Arashloo, M. T., Lavrov, A., Ghobadi, M., Rexford, J., Walker, D., and Wentzlaff, D. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 93–109.

[6] Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proc. CGO* (2019).

[7] Baghdadi, R., Ray, J., Romdhane, M. B., Sozzo, E. D., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019).

[8] Barbette, T., Katsikas, G. P., Maguire, G. Q., and Kostić, D. Rss++: Load and state-aware receive side scaling. In *Proc. CoNEXT* (2019).

[9] Becchi, M., and Crowley, P. A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim. 10*, 1 (Apr. 2013).

[10] Beckmann, O., Houghton, A., Mellor, M., and Kelly, P. H. J. *Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 291–306.

[11] Beckmann, O., Houghton, A., Mellor, M. R., and Kelly, P. H. J. Runtime code generation in c++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation* (2003), hristian Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., vol. 3016, Springer LNCS.

[12] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proc. PPoPP* (2017).

[13] Bhansali, D., et al. Time traveling debugging toolkit. In *Proceedings of the 2nd ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (2006), ACM, pp. 154–163.

[14] Bhatti, S., Brady, E., Hammond, K., and McKinna, J. Domain specific languages (dsls) for network protocols (position paper). In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops* (2009), pp. 208–213.

[15] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *Proc. SIGCOMM* (2014).

[16] Bousse, E., Mayerhofer, T., and Wimmer, M. Domain-level debugging for compiled dsls with the gemoc studio. In *MoDELS* (2017).

[17] Bousse, E., and Wimmer, M. Domain-level observation and control for compiled executable dsls. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2019), pp. 150–160.

[18] Brady, E., and Hammond, K. A verified staged interpreter is a verified compiler. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (New York, NY, USA, 2006), GPCE '06, Association for Computing Machinery, p. 111–120.

[19] Brahmakshatriya, A., and Amarasinghe, S. Buildit: A type based multistage programming framework for code generation in c++. In *Proc. CGO* (2021).

[20] Brahmakshatriya, A., and Amarasinghe, S. Graphit to cuda compiler in 2021 loc: A case for high-performance dsl implementation via staging with buildsl. In *Proc. CGO* (2022).

[21] Brahmakshatriya, A., and Amarasinghe, S. D2x: An extensible contextual debugger for modern dsls. In *Proc. CGO* (2023).

[22] Brahmakshatriya, A., Furst, E., Ying, V., Hsu, C., Hong, C., Ruttenberg, M., Zhang, Y., Jung, D. C., Richmond, D., Taylor, M., Shun, J., Oskin, M., Sanchez, D., and Amarasinghe, S. Taming the zoo: A unified graph compiler framework for novel architectures. In *Proc. ISCA* (2021).

[23] Brahmakshatriya, A., Furst, E., Ying, V. A., Hsu, C., Hong, C., Ruttenberg, M., Zhang, Y., Jung, D. C., Richmond, D., Taylor, M. B., Shun, J., Oskin, M., Sanchez, D., and Amarasinghe, S. Taming the zoo: The unified graphit compiler framework for

novel architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021).

[24] Brahmakshatriya, A., Rinard, C., Ghobadi, M., and Amarasinghe, S. Netblocks: Staging layouts for high-performance custom host network stacks. *Proc. ACM Program. Lang. 8*, PLDI (June 2024).

[25] Brahmakshatriya, A., Zhang, Y., Hong, C., Kamil, S., Shun, J., and Amarasinghe, S. Compiling graph applications for gpus with graphit. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021).

[26] Brahmakshatriya, A., Zhang, Y., Hong, C., Kamil, S., Shun, J., and Amarasinghe, S. Compiling graph applications for gpus with graphit. In *Proc. CGO* (2021).

[27] BuildIt-lang. Einsum dsl implemented with the buildit framework, 2022.

[28] Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., and Agarwal, R. Understanding host network stack overheads. In *Proc. SIGCOMM* (2021).

[29] Calcagno, C., Taha, W., Huang, L., and Leroy, X. Implementing multi-stage languages using asts, gensym, and reflection. In *Proc. GPCE* (2003).

[30] Carette, J., and Kiselyov, O. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program. 76*, 5 (May 2011), 349–375.

[31] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1987).

[32] Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. A domain-specific approach to heterogeneous parallelism. In *Proc. PPoPP* (2011).

[33] Che, S. GasCL: A vertex-centric graph model for GPUs. In *Proc. HPEC* (2014).

[34] Chen, L., Huo, X., Ren, B., Jain, S., and Agrawal, G. Efficient and simplified parallel graph processing over CPU and MIC. In *Proc. IPDPS* (2015).

[35] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proc. OSDI* (2018).

[36] Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., and Seltzer, N. Diderot: A parallel dsl for image analysis and visualization. In *Proc. PLDI* (2012).

[37] Cohen, A., Donadio, S., Garzaran, M.-J., Herrmann, C., Kiselyov, O., and Padua, D. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program. 62*, 1 (Sept. 2006), 25–46.

[38] Colaço, J.-L., Pagano, B., and Pouzet, M. Scade 6: A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (2017).

[39] Collaboration, T. E. H. T. First m87 event horizon telescope results. iv. imaging the central supermassive black hole. *The Astrophysical Journal Letters 875*, 1 (apr 2019), L4.

[40] Committee, D. D. I. F. Dwarf debugging information format version 5, 2017.

[41] community, N. Nginx - a high-performance HTTP server.

[42] Contributors, A. Anydsl: A framework for high-performance domain-specific languages. https://anydsl.github.io/, 2025. Accessed: 2025-08-18.

[43] Contributors, V. Vega: A visualization grammar. https://vega.github.io/vega/, 2025. Accessed: 2025-08-18.

[44] Contributors, V.-L. Vega-lite: A high-level grammar of interactive graphics. https://vega.github.io/vega-lite/, 2025. Accessed: 2025-08-18.

[45] Cook, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[46] Corporation, N. cublas library. https://docs.nvidia.com/cuda/cublas/, 2025. Accessed: 2025-08-18.

[47] Corporation, N. cuda-gdb: Cuda debugger. https://docs.nvidia.com/cuda/cuda-gdb/index.html, 2025. Accessed: 2025-08-18.

[48] Cox, R. Re2: A fast, safe, thread-friendly alternative to backtracking regular expression engines. https://github.com/google/re2, 2009. Accessed: 2025-08-18.

[49] Cox, R. Regular expression matching in the wild. https://swtch.com/~rsc/regexp/regexp3.html, 2010. Accessed: 2025-08-18.

[50] cppreference.com contributors. std::initializer_list. https://en.cppreference.com/w/cpp/utility/initializer_list, 2024. Accessed: 2025-08-18.

[51] Dagum, L., and Menon, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (1998), 46–55.

[52] Davidson, A., Baxter, S., Garland, M., and Owens, J. D. Work-efficient parallel GPU methods for single-source shortest paths. In *Proc. IPDPS* (2014).

[53] Davis, J. C. Rethinking regex engines to address redos. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2019), ESEC/FSE 2019, Association for Computing Machinery, p. 1256–1258.

[54] Davis, T. A. Suitesparse: A suite of sparse-matrix-related packages. https://github.com/DrTimothyAldenDavis/SuiteSparse, 2025.

[55] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., and Vitek, J. Terra: a multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, Association for Computing Machinery, p. 105–116.

[56] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., and Vitek, J. 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.

[57] DeVito, Z., Ritchie, D., Fisher, M., Aiken, A., and Hanrahan, P. First-class runtime generation of high-performance types using exotypes. *SIGPLAN Not. 49*, 6 (June 2014), 77–88.

[58] Di Luna, G. A., Italiano, D., Massarelli, L., Österlund, S., Giuffrida, C., and Querzoni, L. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proc. ASPLOS* (2021).

[59] Dighe, K. Fast multistage compilation of machine learning computation graphs. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2024.

[60] Dow, N. Optimizing scheduling for stream structured programming for streamit. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2025.

[61] Drey, Z., and Teodorov, C. Object-oriented design pattern for dsl program monitoring. In *Proc. SLE* (2016).

[62] Dukkipati, N., Mathis, M., Cheng, Y., and Ghobadi, M. Proportional rate reduction for tcp. In *Proc. SIGCOMM* (2011).

[63] Dunkels, A. uip, 2023.

[64] Engblom, J. A review of reverse debugging. In *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (2015), ACM, pp. 1–10.

[65] Feldman, S. I. Make: A program for maintaining computer programs. https://en.wikipedia.org/wiki/Make_(software), 1979. Accessed: 2025-08-18.

[66] for Standardization, I. O. Iso/iec jtc1/sc22/wg21 n3337: Programming languages – c++ – working draft, january 16, 2012. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf, 2012. Accessed: 2025-08-18.

[67] Foundation, P. S. pdb — the python debugger. https://docs.python.org/3/library/pdb.html, 2025. Accessed: 2025-08-18.

[68] Fu, Z., Liu, Z., and Li, J. Efficient parallelization of regular expression matching for deep inspection. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), pp. 1–9.

[69] Futamura, Y. Partial evaluation of computation process, revisited. *HOSC* (1999).

[70] Gaihre, A., Wu, Z., Yao, F., and Liu, H. XBFS: eXploring runtime optimizations for breadth-first search on GPUs. In *Proc. HPDC* (2019).

[71] Gallo, M., and Laufer, R. ClickNF: a modular stack for custom network functions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 745–757.

[72] Gao, J., Zhai, E., Liu, H. H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., and Yu, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proc. SIGCOMM* (2020).

[73] Ghobadi, M., Yeganeh, S. H., and Ganjali, Y. Rethinking end-to-end congestion control in software-defined networks. In *Proc. HotNets* (2012).

[74] Ghosal, R., Domènech, M. L., Plancher, B., Neuman, S., and Reddi, V. J. Untitled, wip project. Unpublished work, 2025.

[75] Gill, G., Dathathri, R., Hoang, L., Lenharth, A., and Pingali, K. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Proc. Euro-Par* (2018).

[76] Grossman, S., Litz, H., and Kozyrakis, C. Making pull-based graph processing performant. In *Proc. PPoPP* (2018).

[77] Group, A. picotcp, 2012.

[78] Group, F. Freertos-plus-tcp, 2023.

[79] Group, N. Ns2: The network simulator, 2023.

[80] Group, T. K. The opencl™ specification v3.0.19. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html, 2025.

[81] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[82] Han, W., Mawhirter, D., Wu, B., and Buland, M. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proc. PACT* (2017).

[83] Harish, P., and Narayanan, P. Accelerating large graph algorithms on the GPU using CUDA. In *Proc. HIPC* (2007).

[84] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

[85] Hong, C., Sukumaran-Rajam, A., Kim, J., and Sadayappan, P. Multigraph: Efficient graph processing on GPUs. In *Proc. PACT* (2017).

[86] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-Marl: A DSL for easy and efficient graph analysis. In *Proc. ASPLOS-XVII* (2012).

[87] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. Accelerating CUDA graph algorithms at maximum warp. In *Proc. PPoPP* (2011).

[88] Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* (2019).

[89] Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* (2019).

[90] Hüni, H., Johnson, R., and Engel, R. A framework for network protocol software. *SIGPLAN Not.* (1995).

[91] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[92] Hwang, J., Cai, Q., Tang, A., and Agarwal, R. TCP = RDMA: Cpu-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 127–140.

[93] Inc., K. Cmake: A cross-platform makefile generator. https://cmake.org/, 2025. Accessed: 2025-08-18.

[94] Jang, J., and Adib, F. Underwater backscatter networking. In *Proc. SIGCOMM* (2019).

[95] Jeffrey, M. C., Subramanian, S., Yan, C., Emer, J., and Sanchez, D. A scalable architecture for ordered parallelism. In *Proc. MICRO-48* (2015).

[96] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 489–502.

[97] Jiang, X., Wang, H., Chen, Y., Wu, Z., Wang, L., Zou, B., Yang, Y., Cui, Z., Cai, Y., Yu, T., Lv, C., and Wu, Z. Mnn: A universal and efficient inference engine, 2020.

[98] Johnson, S. C. Yacc: Yet another compiler compiler. https://en.wikipedia.org/wiki/Yacc, 1975. Accessed: 2025-08-18.

[99] Kalia, A., Kaminsky, M., and Andersen, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 1–16.

[100] Katai-IO. Katai struct compiler, 2022.

[101] Khorasani, F., Gupta, R., and Bhuyan, L. N. Scalable SIMD-efficient graph processing on GPUs. In *Proc. PACT* (2015).

[102] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. CuSha: Vertex-centric graph processing on GPUs. In *Proc. HPDC* (2014).

[103] Kildall, G. A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1973).

[104] Kim, M.-S., An, K., Park, H., Seo, H., and Kim, J. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proc. SIGMOD* (2016).

[105] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. In *Proc. OOPSLA* (2017).

[106] Kjolstad, F., Kamil, S., Ragan-Kelley, J., Levin, D. I. W., Sueda, S., Chen, D., Vouga, E., Kaufman, D. M., Kanwar, G., Matusik, W., and Amarasinghe, S. Simit: A language for physical simulation. *ACM Trans. Graph.* (2016).

[107] Kjolstad, F., Kamil, S., Ragan-Kelley, J., Levin, D. I. W., Sueda, S., Chen, D., Vouga, E., Kaufman, D. M., Kanwar, G., Matusik, W., and Amarasinghe, S. Simit: A language for physical simulation. *ACM Trans. Graph.* (2016).

[108] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst.* (2000).

[109] Kumar, A. Automatic conversion of c and c++ programs to the buildit multi-stage programming framework. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2025.

[110] Kumar, G., Dukkipati, N., Jang, K., Wassel, H. M. G., Wu, X., Montazeri, B., Wang, Y., Springborn, K., Alfeld, C., Ryan, M., Wetherall, D., and Vahdat, A. Swift: Delay is simple and effective for congestion control in the datacenter. SIGCOMM '20, Association for Computing Machinery, pp. 514–528.

[111] Kurose, J. F., and Ross, K. W. *Computer Networking: A Top-Down Approach*, 5th ed. Addison-Wesley Publishing Company, USA, 2009.

[112] Lam, M. S., Guo, S., and Seo, J. SociaLite: Datalog extensions for efficient social network analysis. In *Proc. ICDE* (2013).

[113] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., and Shi, Z. The quic transport protocol: Design and internet-scale deployment. In *Proc. SIGCOMM* (2017), SIGCOMM '17.

[114] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. https://en.wikipedia.org/wiki/LLVM, 2004. Accessed: 2025-08-18.

[115] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J. A., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *Proc. CGO* (2021).

[116] Lewis, B. Debugging backwards in time. *arXiv preprint cs/0310016* (2003).

[117] Li, D., Chakradhar, S., and Becchi, M. GRapid: A compilation and runtime framework for rapid prototyping of graph applications on many-core processors. In *Proc. ICPADS* (2014).

[118] Li, H., Wu, C., Sun, G., Zhang, P., Shan, D., Pan, T., and Hu, C. Programming network stack for middleboxes with rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 551–570.

[119] Liu, B., and Huang, J. D4: Fast concurrency debugging with parallel differential analysis. In *Proc. PLDI* (2018).

[120] Liu, H., and Huang, H. H. Enterprise: breadth-first graph traversal on GPUs. In *Proc. SC* (2015).

[121] Liu, H., and Huang, H. H. SIMD-x: Programming and processing of graph algorithms on GPUs. In *Proc. USENIX ATC* (2019).

[122] Marinos, I., Watson, R. N. M., and Handley, M. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2013), HotNets-XII, Association for Computing Machinery.

[123] Martin, E. Ninja: A small build system with a focus on speed. https://ninja-build.org/, 2012. Accessed: 2025-08-18.

[124] Masiero, R., Azad, S., Favaro, F., Petrani, M., Toso, G., Guerra, F., Casari, P., and Zorzi, M. Desert underwater: An ns-miracle-based framework to design, simulate, emulate and realize test-beds for underwater network protocols. In *2012 Oceans - Yeosu* (2012), pp. 1–10.

[125] Menon, A., and Zwaenepoel, W. Optimizing tcp receive performance. In *USENIX 2008 Annual Technical Conference* (USA, 2008), ATC'08, USENIX Association, pp. 85–98.

[126] Merrill, D., Garland, M., and Grimshaw, A. High-performance and scalable GPU graph traversal. *ACM Trans. Parallel Comput.* (2015).

[127] Meta. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, July 2024. Accessed: 2025-08-14.

[128] Microsoft. Windbg: A multipurpose debugger for windows. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-reference, 2025. Accessed: 2025-08-18.

[129] Mitrovska, T. Implementing breeze - a high-performance regular expression library using code generation with buildit. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2023.

[130] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. Timely: Rtt-based congestion control for the datacenter. In *Proc. SIGCOMM* (2015).

[131] Nasre, R., Burtscher, M., and Pingali, K. Data-driven versus topology-driven irregular computations on GPUs. In *Prof. IPDPS* (2013).

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

[133] Nodehi Sabet, A. H., Qiu, J., and Zhao, Z. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proc. ASPLOS-XXIII* (2018).

[134] OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, Aug. 2025. Accessed: 2025-08-14.

[135] Pai, S., and Pingali, K. A compiler for throughput optimization of graph algorithms on GPUs. In *Proc. OOPSLA* (2016).

[136] Palsberg, J., and Jay, C. B. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference* (USA, 1998), COMPSAC '98, IEEE Computer Society, p. 9–15.

[137] Parr, T. Antlr: Another tool for language recognition. https://en.wikipedia.org/wiki/ANTLR, 2025. Accessed: 2025-08-18.

[138] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.

[139] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library, 2019.

[140] Paxson, V. Flex: Fast lexical analyzer generator. https://en.wikipedia.org/wiki/Flex_(lexical_analyzer_generator), 2025. Accessed: 2025-08-18.

[141] Peng, Z., Powell, A., Wu, B., Bicer, T., and Ren, B. GraphPhi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proc. PACT-26* (2018).

[142] Peverelli, F., Brahmakshatriya, A., Conficconi, D., Santambrogio, M. D., and Amarasinghe, S. MARCH: Multi-staging ARray compiler for high-level synthesis. Preprint, 2025. Available upon request.

[143] Project, L. Lldb: A next-generation, high-performance debugger. https://lldb.llvm.org/, 2025. Accessed: 2025-08-18.

[144] Project, P. Pcre2: Perl-compatible regular expressions. https://github.com/PCRE2Project/pcre2, 2025.

[145] Project Gutenberg. The complete works of mark twain. https://www.gutenberg.org/files/3200/, 2021. Accessed: 2025-08-18.

[146] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. 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* (2013).

[147] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. PLDI* (2013).

[148] Ray, J. *A Universal Tensor Abstraction and its Application to and Implementation within Block-Based Compression*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2023.

[149] Research, A. Graphviz: Graph visualization software. https://graphviz.org/, 2025. Accessed: 2025-08-18.

[150] Robison, A. D., and Leiserson, C. E. Cilk plus. In *Programming Models for Parallel Computing*, P. Balaji, Ed. MIT Press, 2015, pp. 313–330.

[151] Rompf, T., and Odersky, M. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (2010).

[152] Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D., and Richtarik, P. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association.

[153] Saraff, D. Custom protocols for efficient utilization of bandwidth-constrained networks. Meng thesis, Imperial College London, London, United Kingdom, 2024.

[154] Savidis, A., and Tsiatsianas, V. Implementation of live reverse debugging in lldb. *arXiv preprint arXiv:2105.12819* (2021).

[155] Shajii, A., Numanagić, I., Baghdadi, R., Berger, B., and Amarasinghe, S. Seq: A high-performance language for bioinformatics. In *Proc. OOPSLA* (2019).

[156] Sheard, T., and Jones, S. P. Template meta-programming for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell* (October 2002), ACM.

[157] Shi, X., Luo, X., Liang, J., Zhao, P., Di, S., He, B., and Jin, H. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *TKDE* (2017).

[158] Shun, J., and Blelloch, G. E. Ligra: A lightweight graph processing framework for shared memory. In *Proc. PPoPP* (2013).

[159] Shun, J., Dhulipala, L., and Blelloch, G. E. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Proc. DCC* (2015).

[160] Soman, J., Kishore, K., and Narayanan, P. A fast GPU algorithm for graph connectivity. In *Proc. IPDPSW* (2010).

[161] Stallman, R. M., Pesch, R., and Shebs, S. Debugging with gdb: The gnu source-level debugger. https://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_toc.html, 2002. Accessed: 2025-08-18.

[162] Stanford Compiler Group. The builder library, a tool to construct or modify suif code within the suif compiler, 1994.

[163] Steele, G. L., and Sussman, G. J. Scheme: An interpreter for extended lambda calculus, 1975. AI Memo, December 1975.

[164] Sun, J., Vandierendonck, H., and Nikolopoulos, D. S. GraphGrind: Addressing load imbalance of graph partitioning. In *Proc. ICS* (2017).

[165] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. GraphMat: High performance graph analytics made productive. *VLDB Endow.* (2015).

[166] Swadi, K., Taha, W., Kiselyov, O., and Pasalic, E. A monadic approach for avoiding code duplication when staging memoized functions. In *Proc. PEPM* (2006).

[167] Syzov, D., Kachan, D., Karpov, K., Mareev, N., and Siemens, E. Custom udp-based transport protocol implementation over dpdk. vol. 7.

[168] Taha, W. A gentle introduction to multi-stage programming. In *Proc. Domain-Specific Program Generation*. 2004.

[169] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *SIGPLAN Notices* (1997).

[170] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *SIGPLAN Not.* (1997).

[171] Tantau, T. Tikz: A latex package for creating graphics. https://tex.stackexchange.com/questions/144343/citing-tikz-and-other-packages, 2025. Accessed: 2025-08-18.

[172] Taylor, I. L. libbacktrace: A c library for symbolic backtraces, 2012.

[173] Team, D.-V. Deepseek-v3 technical report, 2025.

[174] Team, G. G. . Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[175] TensorFlow Development Team. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[176] Thies, W., Karczmarek, M., and Amarasinghe, S. P. Streamit: A language for streaming applications. In *Lecture Notes in Computer Science*, vol. 2401. Springer, 2002, pp. 179–196.

[177] Thompson, K. Programming techniques: Regular expression search algorithm. *Commun. ACM 11*, 6 (June 1968), 419–422.

[178] Tillet, P., Kung, H.-T., and Cox, D. D. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)* (2019), ACM, pp. 10–19.

[179] van de Geijn, R., and Goto, K. *BLAS (Basic Linear Algebra Subprograms)*. Springer US, Boston, MA, 2011, pp. 157–164.

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

[181] Varda, K. Protocol buffers: Google's data interchange format. Tech. rep., Google, 6 2008.

[182] Viiret, J. Hyperscan: Performance analysis of hyperscan with hs-bench. https://www.intel.com/content/www/us/en/collections/libraries/hyperscan/performance-analysis-hyperscan-hsbench.html, 2019. Accessed: 2025-08-18.

[183] Vilk, J., Berger, E. D., Mickens, J., and Marron, M. Mcfly: Time-travel debugging for the web. *arXiv preprint arXiv:1810.11865* (2018).

[184] Visser, E. *WebDSL: A Case Study in Domain-Specific Language Engineering*. 2008.

[185] Vora, K., Gupta, R., and Xu, G. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proc. ASPLOS-XXII* (2017).

[186] Wang, H., Geng, L., Lee, R., Hou, K., Zhang, Y., and Zhang, X. SEP-Graph: Finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proc. PPoPP* (2019).

[187] Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. P4fpga: A rapid prototyping framework for p4. In *Proc. SOSR* (2017).

[188] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 631–648.

[189] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the GPU. In *Proc. PPoPP* (2016).

[190] Wei, G., Chen, Y., and Rompf, T. Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming. In *Proc. OOPSLA* (2019).

[191] Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., and Taha, W. Mint: Java multi-stage programming using weak separability. *SIGPLAN Not.* (2010).

[192] Williams, T., and Kelley, C. Gnuplot 4.5: An interactive plotting program. http://gnuplot.info, 2011. Accessed: 2025-08-18.

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

[194] Wu, H., Gray, J., and Mernik, M. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience* (2008).

[195] Wu, H., Gray, J. G., and Mernik, M. Debugging domain-specific languages in eclipse.

[196] Yuki, T., Gupta, G., Kim, D., Pathan, T., and Rajopadhye, S. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing* (Berlin, Heidelberg, 2013), H. Kasahara and K. Kimura, Eds., Springer Berlin Heidelberg, pp. 17–31.

[197] Zhang, K., Chen, R., and Chen, H. NUMA-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[198] Zhang, Y., Brahmakshatriya, A., Chen, X., Dhulipala, L., Kamil, S., Amarasinghe, S., and Shun, J. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (2020).

[199] Zhang, Y., Brahmakshatriya, A., Chen, X., Dhulipala, L., Kamil, S., Amarasinghe, S., and Shun, J. Optimizing ordered graph algorithms with graphit. In *Proc. CGO* (2020).

[200] Zhang, Y., Kiriansky, V., Mendis, C., Amarasinghe, S., and Zaharia, M. Making caches work for graph analytics. In *Proc. BigData* (2017).

[201] Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S. Graphit: a high-performance graph dsl. *Proc. ACM Program. Lang. 2*, OOPSLA (2018).

[202] Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S. Graphit: A high-performance graph dsl. In *Proc. OOPSLA* (2018).