D2X: An eXtensible conteXtual Debugger for Modern DSLs

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

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

Abstract

Compiled Domain Specific Languages are taking over various high-performance domains because of their ability to exploit the domain knowledge and apply optimizations that produce the most specialized code. A lot of research has gone into making DSLs more performant and easy to prototype. But the Achilles heel for DSLs is still the lack of debugging support that provides an end-to-end picture to the user and improves the productivity of both the DSL designer and the end-user. Conventional techniques extend the compilers, the debugging information format, and the debuggers themselves to provide more information than what the debugger can provide when attached to the generated code. Such an approach quickly stops scaling as adding extensions to large and complex debuggers hampers DSL designer productivity. We present D2X, a DSL debugging infrastructure that works with most standard debuggers without any modifications and is easily extensible to capture all the domain-specific information the end-user cares about. We show that we can add debugging support to the state-of-the-art graph DSL GraphIt with as little as 1.4% changes to the compiler code base. We also apply our techniques to a meta-programming DSL framework BuildIt so that any DSLs built on top of BuildIt get debugging support without any modifications further boosting the productivity of future DSL designers.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Keywords: DSLs, compilers, debuggers

ACM Reference Format:
Ajay Brahmakshatriya and Saman Amarasinghe. 2023. D2X: An e**X**tensible conte**X**tual **D**ebugger for Modern **D**SLs. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3579990.3580014

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02.
https://doi.org/10.1145/3579990.3580014

1 Introduction

Recent years have seen a significant uptick in the use of computation for solving problems in various emerging and already existing areas of research. From using DNA sequencing for COVID-19 vaccine research to imaging black holes and distant stars [15], many fields are relying on performance-intensive computations to produce better and faster results. Even in areas such as data mining, machine learning, and scientific simulations that have long relied on computation, problem sizes have increased by up to 300,000x and are demanding more performance-efficient software and hardware stacks[21, 24]. This influx of new problem domains requiring high performance has pushed compilers and programming language research towards more domain-specific languages (DSLs) for their ability to better specialize the optimizations to the problem domain. DSLs like Halide [22], TACO [2], Taichi [16] are being developed and adopted for these domains while frameworks like MLIR [19] and BuildIt [5, 7] are focusing on streamlining DSL development making it accessible to domain experts with little to no compilers knowledge.

While a significant amount of research has been done in improving the performance of these DSLs, providing better-debugging facilities is often overlooked. In cases where DSL compilers generate low-level C, C++, or CUDA code, the end-users have to resort to attaching debuggers like GDB, LLDB, or CUDA-gdb to the generated code. This generated code is often very complicated and mangled and interspersed with other user and library code. This makes it not human readable, and has little correlation to the input code written by the end-user. This is because of the transformation passes that eliminate, combine or create new operators and variables. Apart from this, the DSL compilers make many domain-specific optimization decisions and transformations to the code. Because these decisions are not exposed to the end-user, they often have to guess how the input code was modified before it was generated. This is a challenge for both correctness and performance debugging.

Extending existing debugging infrastructure to accommodate the needs of DSLs is often a difficult and time-consuming task. Low-level language compilers encode debugging information as binary data in formats like DWARF [12] or PDB which are hard to understand and extend. At the same time, modifications would have to be made to the debugger itself in the form of plugins or modifying the code of the debugger itself. Such an approach is taken by the developers of the CUDA-gdb, where the ability to debug GPU kernels is added to the existing GDB infrastructure. Not only would these modifications be not portable to other platforms and debuggers, but these efforts would also take away valuable research and prototyping time from DSL developers.

We propose D2X (pronounced as Detox), an extensible and contextual debugging framework that allows DSL developers to encode semantic debugging details from the DSL compiler alongside the generated code. D2X also provides a runtime API that can be used to access this debugging information from existing debuggers in a portable way without any modifications. Besides providing basic debugging functionality like listing source location and variables and managing breakpoints, D2X allows the DSL designers to arbitrarily extend the debugging information to expose any combination of compiler internal state and runtime information. D2X presents itself as an easy-to-use C++ library that easily integrates into any DSL compiler code base. We demonstrate our techniques by applying D2X to the state-of-the-art graph DSL compiler GraphIt to debug the generated CPU parallel code while requiring about 1.4% changes to the code base. We also apply our techniques to the DSL compiler framework BuildIt 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. D2X thus increases the productivity of not only the end-user but also current and future DSL designers.

In the rest of the paper, we will explain the debugging requirements of DSLs and the challenges of using existing infrastructure (Section 2). Next we will explain the overview of our system D2X (Section 3) and the technical details of its implementation (Section 4). Finally, we will talk about the two case studies with the GraphIt DSL compiler and the BuildIt DSL framework and how our work simplifies DSL development and use. We will also quantify the amount of effort required to add debugging support for these DSLs in terms of lines of code modified (Section 5).

2 Challenges

DSL compilers have grown in complexity over years and present some unique challenges when it comes to building debugging infrastructure. In this section, we will discuss these challenges and some examples from popular DSLs.

2.1 Disconnect in Source and Generated Code

Domain Specific Language compilers often 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 [8, 29, 30], SIMIT [18] and Diderot [11]. Other DSLs like Halide [22], Taichi [16], and Tiramisu [2] embed themselves into host languages like python or C++ for providing a familiar interface. Some DSLs like TACO [17] sometimes provide both these interfaces. The input code is passed through a series of analysis and transformation passes and combined with scheduling inputs before generating the final code. Often the code is parallelized, vectorized, or moved to GPU as specialized kernels. Naturally, the generated code has very little resemblance to the input code. For example, in the popular graph DSL GraphIt, 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 iteration 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.

```
1 func updateEdge(s: Vertex, d: Vertex)
2   nrank[d] += orank[s]
3 end
4 func main()
5 #s1# edges.apply(updateEdge) // PUSH Schedule
6 #s2# edges.apply(updateEdge) // PULL Schedule
7 end
```

Figure 1. GraphIt algorithm input with same User Defined Function (UDF) `updateEdge` used with two operators one with PUSH schedule applied and the other with PULL.

Figure 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 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 the multi-stage programming framework BuildIt, 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 is simply not present.

2.2 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

```
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 2. Generated code for the GraphIt input in Figure 1. The same UDF is generated into two separate versions suited for the two call sites

example, in the Sparse Tensor DSL TACO, individual sparse tensors are stored in one or more formats such as CSR, COO, BCSR based on the operation being performed. 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 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.

2.3 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 DSLs 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 [25] 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-information is neither present in the source code nor the generated binary. Displaying this type of information would require extending the debugger and the binary format used to encode the debugging information. The multistage programming framework BuildIt also has a similar problem where the state of the first stage variables are not visible in the generated code but are 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 [12] the popular debugging format used on Linux has a standard that spans 459 pages, and the complexity required to understand is beyond the scope of a typical domain 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

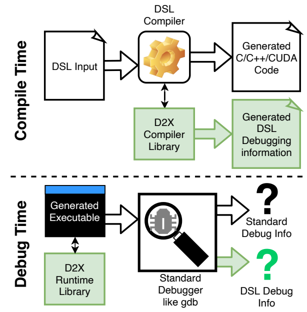

Figure 3. The overall system overview for the D2X compiler and runtime extensions. Additional components added/generated by D2X are in green.

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.

All the above-mentioned issues guide our design of D2X. We 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 by to display semantic information.

3 System Overview

D2X is a C++ library that comprises of two parts – the D2X compiler library (D2X-C) and the D2X runtime library (D2X-R). Figure 3 shows the overall working of D2X. We choose C++ for our framework because most high-performance DSL compilers are written in C++ and the generated code is often compatible with C++.

3.1 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 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. We will discuss the amount of modification required for the DSLs we modified in Section 5.

3.2 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

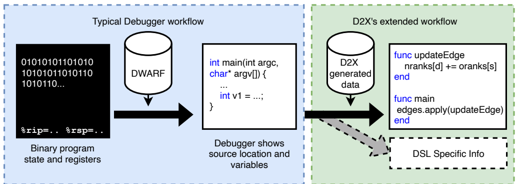

Figure 4. Two-stage mapping of source information enabled with D2X. Left (blue) shows typical mapping from binary state to source using DWARF. Right (green) shows mapping from generated source to DSL input using data generated by D2X-C.

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. D2X-R's implementation is explained in Section 4.

3.3 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.

4 Implementation

Most high-performance DSL developers and end-users are familiar with using low-level debugging tools like GDB, LLDB, and WinDbg. Instead of techniques that build ad-hoc interfaces for debugging, we make a design decision of using these commonly used debuggers to make adoption easier.

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 uses the tables D2X-C generates alongside the source. Figure 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. We will discuss the implementation of D2X-C to see what data structures are generated during compilation.

4.1 Implementation of D2X-C

As explained in Section 3, D2X-C is available as a C++ library that can be directly used inside the DSL compiler. The developer calls the D2X-C API to encode DSL related debug information. The complete API is listed in Table 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 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 to not misalign the generated code and the debug information.

The examples in Section 2 show that the source location in the DSL is more elaborate than a single line of source code. As a result, we designed D2X to allow for a sequence 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.

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 have 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. Section 5 shows examples of data flow information encoded in the GraphIt compiler. 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 string and produces the value as a string. We use BuildIt's multi-stage execution to supply this lambda. Table 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.

Table 1. Table showing the API functions from the D2X compiler API (D2X-C). Argument names shown in [] are optional. The `rt::string` type is a BuildIt dynamic type for handling strings at runtime (`dyn_var<string>`).

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 sope 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

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 mechanism 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. Section 5 shows 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 generates them into the C++ file specified. The D2X-R functions access this source and variable information by reading the generated arrays. This is explained in Section 4.2

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 embedded in C++ and use the compiler in the form of a library and do not have a separate parser.

4.2 Implementation of D2X-R

As explained before, we designed D2X to work with commonly used debuggers because both DSL developers and end-users that 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, we want to enable the DSL designers to add more features to the implementation as they wish. These constraints motivate a novel design for our debugger runtime.

All the modern debuggers that we inspected (GDB, LLDB, and WinDbg) implement a feature where arbitrary functions from the program 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. We exploit this feature of the debuggers to implement extensions to the debugger. The D2X runtime library (D2X-R) links functions into the executable that have a well-defined interface and allows 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 we implement are just calls to these functions. These functions use the passed instruction pointer and stack pointer to identify where the execution 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 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.

As mentioned in Section 3, the D2X-R also supplies a set of macros that make it easy to call these functions. Figure 5 also shows how these macros are invoked.

The main benefit of our approach is that the entire implementation of the debugger extension is simply written as C++ code that is linked into the executable, thus requiring

```
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 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.

no modification or plugins to the debugger itself. This also makes our approach very portable and easy to extend as long as the debugger supports calling functions from the executable. Our approach is also strictly better than an alternative approach where the DSL compiler would just insert source annotations into the generated code that the backend compiler would understand. Such an approach would allow only for a one-to-one mapping between the DSL input and the generated code and would be severely limited by what the debugger can do with the source information. Our extensions allow the DSL developers to encode arbitrary data and process it in any way they like in the debugger.

D2X also works with multi-threaded programs which is typical for most high-performanace DSLs without any modifications as long as the debugger has the ability to pause individual threads.

Table 2 shows all the commands that can be invoked from the D2X-R API and their description. We 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. Remember 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 with the `xframe` command similar to the typical GDB command `frame`. Our 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. Examples discussed in Section 5 with applications.

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. Our technique of invoking D2X-R functions from the debugger that print information is not enough to manage breakpoints. We need the ability to invoke debugger commands as a result of the D2X-R API call. To enable this, we leverage 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, we now invoke it as `eval "%s", d2x_runtime::command_xbreak($rip, ...)`. The `command_xbreak` functions takes as parameters a source location and 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 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.

4.3 DSL Specific Extensions to D2X

Besides being easy to use for both DSL designers and end-user, 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

Table 2. Command macros that can be invoked from the debugger and their descriptions. The optional arguments are shown in `[]`. Each of these macros invoke the corresponding D2X-R API function with the `$rip` and `$rsp` as parameters.

Command Macros	Description
`xbt`	Displays the extended stack associated with the current execution stack frame. 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.

and would require messing with the large code bases of the debuggers, hurting the productivity of the DSL designers.

5 Case Studies

This section details our experiences with applying D2X's debugging capabilities to the graph DSL GraphIt and BuildIt a framework that makes it easy for non-compiler experts to implement DSL compilers.

5.1 Debugging Capabilities for GraphIt

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. As explained in Section 2, 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. Since GraphIt is a popular DSL that generates state-of-the-art high-performance code for graph applications, it is critical that we improve the debugging support for improving end-user productivity. We will explain two main capabilities that we add to GraphIt and how they appear in the debugger.

Source locations in GraphIt. 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 for us to support D2X extensions without modifying the parser. We propagate the line numbers from the parser through all the mid-end passes to the code generation phase that 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 (UDF), we also insert 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 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. As explained before, 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. We implement support for debugging `vertexsets` with the help of a `rtv_handler`. Figure 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 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 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.

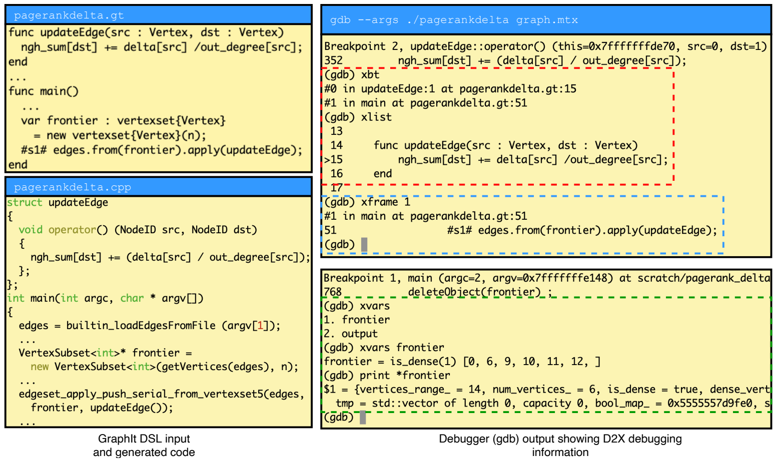

Figure 6. The GraphIt DSL input and the generated code for PagerankDelta and the view from the debugger (GDB). The red box shows the extended stack and the listing using `xbt` and `xlist` commands. The blue box shows the UDF calling context using `xframe`. The green box shows the `vertexset` objects and the output from the `rtv_handler` using `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. Definition of the `rtv_handler` to display a `vertexset`. Notice the check on the format and the serialization based on it.

Table 3. The number of lines of code changed in GraphIt and number of lines of code required for the implementation of D2X. We add support for contextual debugging to GraphIt by changing merely 1.4% lines of code.

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**

Furthermore, Table 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.

5.2 BuildIt DSL Framework

For the next application, we look at the BuildIt DSL framework [5, 7]. BuildIt is a DSL framework that uses multi-stage programming in C++ to enable rapid prototyping of high-performance DSLs with little to no compiler knowledge. This approach of using multi-stage programming allows domain experts to write library-like code, specialize it based on compile time inputs, perform analysis using static variables and generate code for CPUs and GPUs. The BuildIt framework has shown that the entire GraphIt DSL can be implemented for the GPU architecture using only 2021 lines of code [7] which is a huge step toward developer productivity. With this case study, we add D2X support to all the code generated from BuildIt which enables debugging capabilities for all DSLs built on top of BuildIt further boosting the productivity of not only the end-users but also the DSL designers.

The key idea behind BuildIt is that the program is written in multiple stages. BuildIt introduces two new template types `static<T>` and `dyn<T>`. All code written using `static<T>` is completely evaluated in the first stage while the expressions, variables, and control flow written with `dyn<T>` are generated as it is in the second stage code. This second stage code is then further compiled and executed. There is an obvious disconnect in the source the developer writes vs the second stage code that would be executed with a debugger. Providing source locations from the first stage code would allow for end-to-end debugging that would span multiple stages. Just like the previous GraphIt DSL, each generated line of code can be a result of an entire call stack in the first stage as opposed to a single line. Furthermore, the `static<T>` variables are completely erased from the generated code, which means that these variables are not visible when the user tries to debug the generated code. But these variables definitely have an effect on the generated code because 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. 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 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. We use D2X-C's helpful `self_source_loc` util API (described in Section 4) to obtain 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<T>` variable follows a similar procedure. Each statement in the generated code is already attached with a snapshot of all the live `static<T>` variables. We simply serialize each `static<T>` variable from the snapshot into a string and encode it as an `xvar` using the D2X-C API.

Figure 8 shows the input and generated code for a power function using repeated squaring. Figure 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<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.

Debugging DSLs with BuildIt. One of the main functions of the BuildIt framework is to rapidly prototype high-performance DSLs. Developers build library-like abstractions on top of BuildIt and specialize it using `static<T>` inputs before code generation. Our techniques allow the end-users to peek into the state of the analyses and specializations performed inside the DSL compiler as well as observe the embedded DSL input. We take a simple Einsum expression DSL compiler on top of BuildIt [9] that is available on the BuildIt website and repository. 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.

```
// First stage BuildIt code
dyn<int> power_f(dyn<int> base, static<int> exponent) {
  dyn<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 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<int>` `exponent` is completely erased from the generated code but produces a sequence of statements.

```
(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 9. Debugger (GDB) output for the code generated in Figure 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` command show the first stage source and the state of the `static<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 10. An input for Einsum lang implemented on top of BuildIt that initializes a vector performs matrix-vector multiplication. 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 11. The D2X output from the debugger for the program in Figure 10. The `xbt` command shows the steps inside the DSL implementation and the `xvars` command shows the details of the analysis stored as `static<T>`.

Table 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.

Component	Lines of C++ Code
BuildIt DSL compiler framework	7,035
Delta for adding D2X support	428 (+407/-21)
BuildIt percentage change	**6.1%**

The implementation also performs constant propagation in tensors by the use of static variables. We use D2X and the generated debugging information to inspect this internal state. We test the embedded DSL on an input program (Figure 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<T>` variables. Figure 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 we did not make even a single line of change in 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 4 shows the total number of lines and the lines changed.

6 Related Works

High-performance DSLs are gaining popularity across various domains like graph computations [8, 29, 30], tensors [16, 17], polyhedral compilation [2], genome sequencing [25], image processing and analysis [11, 22], machine learning [1, 10] among others. Recently, there has been a focus on building infrastructure support for prototyping high-performance DSLs with ease [5, 7, 19, 23]. Most of these DSLs focus primarily on the maximum achievable performance and very little on ease of debugging. These DSLs provide support for source-level debugging but lack support for DSL specific contextual debugging and any kind of extensibility.

Some research has been focused on building new interfaces for debugging DSLs [3, 4, 14, 26–28], 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 [20] but typically have a narrow focus like concurrency and do not adapt to the need of upcoming DSLs. D2X on the other hand simply generates regular C++ data structures and code that can be used with any debugger that supports calling functions from the executables. This makes our approach not only portable but also easily extensible. The extra C++ code that we generate is never executed till the end-user invokes a debugging command.

General purpose language debuggers have been around for a while and rely on the compiler generating standardized debugging formats like DWARF [12] 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 [13] but these guarantees are very hard to extend to domain-specific extensions. We show the robustness of D2X by applying it to a high-performance state-of-the-art DSL and an easy-to-use DSL framework which extends the benefits of our approach to future DSLs.

7 Conclusion

In this paper, we present D2X which is a portable end-to-end debugging framework for DSLs and provides rich debugging information that can be easily extended to fit the needs of upcoming DSLs with ease. We apply D2X's debugging techniques to GraphIt and the easy-to-use DSL framework BuildIt. D2X thus greatly increases productivity for not only the end-user but also the designers.

Acknowledgment

This work was supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; DARPA under Awards HR0011-18-3-0007 and HR0011-20-9-0017; NSF under award CCF-2107244; and Intel/NSF award CCF-2217064.

Data Availability Statement

D2X and all its components are available open-source [6] under the MIT License. All of the source code for D2X (D2X-C and D2X-R) can be found under the D2X directory. The source code for the modified compiler for the graph DSL, GraphIt and BuildIt can be found under the `graphit` and `buildit` directories respectively. The `README` in the repository contains instructions to build all dependencies and applications and reuse the framework for more applications.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proc. CGO*.

[3] Erwan Bousse, Tanja Mayerhofer, and Manuel Wimmer. 2017. Domain-level Debugging for Compiled DSLs with the GEMOC Studio. In *MoDELS*.

[4] Erwan Bousse and Manuel Wimmer. 2019. Domain-Level Observation and Control for Compiled Executable DSLs. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 150–160.

[5] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A type based multistage programming framework for code generation in C++. In *Proc. CGO*.

[6] Ajay Brahmakshatriya and Saman Amarasinghe. 2022. *Artifacts for the CGO23 paper: D2X: An eXtensible conteXtual Debugger for modern DSLs*. https://doi.org/10.5281/zenodo.7459640

[7] Ajay Brahmakshatriya and Saman Amarasinghe. 2022. GraphIt to CUDA Compiler in 2021 LOC: A Case for High-Performance DSL Implementation via Staging with BuilDSL. In *Proc. CGO*.

[8] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPUs with GraphIt. In *Proc. CGO*.

[9] BuildIt-lang. 2022. Einsum DSL implemented with the BuildIt framework. https://buildit.so/tryit/?sample=einsum

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. OSDI*.

[11] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proc. PLDI*.

[12] DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format Version 5. https://dwarfstd.org

[13] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries. In *Proc. ASPLOS*.

[14] Zoé Drey and Ciprian Teodorov. 2016. Object-Oriented Design Pattern for DSL Program Monitoring. In *Proc. SLE*.

[15] The Event Horizon Telescope Collaboration et. al. 2019. First M87 Event Horizon Telescope Results. IV. Imaging the Central Supermassive Black Hole. *ApJL* (2019).

[16] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* (2019).

[17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. In *Proc. OOPSLA*.

[18] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* (2016).

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proc. CGO*.

[20] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proc. PLDI*.

[21] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. PLDI*.

[23] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs. *Electronic Proceedings in Theoretical Computer Science* (2011).

[24] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* (2020).

[25] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. 2019. Seq: A High-Performance Language for Bioinformatics. In *Proc. OOPSLA*.

[26] Eelco Visser. 2008. *WebDSL: A Case Study in Domain-Specific Language Engineering*.

[27] Hui Wu, Jeff Gray, and Marjan Mernik. 2008. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience* (2008).

[28] Hui Wu, Jeffrey G. Gray, and Marjan Mernik. 2004. Debugging Domain-Specific Languages in Eclipse.

[29] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *Proc. CGO*.

[30] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. In *Proc. OOPSLA*.