School of Computation, Information and Technology — Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Efficient Generation of Debugging Information

Markus Gschoßmann

School of Computation, Information and Technology — Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Effiziente Erzeugung von Debugging-Information

Efficient Generation of Debugging Information

Author:	Markus Gschoßmann
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Alexis Engelke
Submission Date:	29.08.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Markus Gschoßmann

Acknowledgments

First, I would like to thank my advisor Dr. Alexis Engelke for providing an incredible amount of feedback to his students, so much in fact, that we sometimes disturbed his schedule due to long conversations.
Furthermore, I would like to thank Tobias Schwarz for being my unofficial second advisor, especially in the beginning.
Finally, I also thank Ryan Stinnett and Stephen Kell for providing support on their program.

Abstract

Programmer productivity is significantly affected by long compilation times, to which the high number of passes in typical compiler back-ends contribute. TPDE provides an alternative LLVM compiler back-end, which achieves substantial speedups by reducing the number of passes. However, emission of debugging information is not yet supported, preventing adoption for development purposes.

In this thesis, we extend TPDE with support for efficiently emitting DWARF5 debugging information based on metadata in the input LLVM-IR. In particular, we derive efficient intermediate data structures and prioritize fast emission over short encodings.

When compiling SPECint 2017 without optimizations, generating debugging information slows down TPDE by 45% on average. Compared to LLVM, TPDE with debugging information compiles SPECint 2017 about 7 times faster.

Contents

1 Introduction **1**

2 Technical Background **3**
- 2.1 Debugging Information . . . 3
- 2.2 DWARF . . . 3
- 2.3 LLVM . . . 4

3 Line Table **5**
- 3.1 Representation in DWARF . . . 5
  - 3.1.1 Fields . . . 5
  - 3.1.2 Compression . . . 6
- 3.2 Encoding in LLVM IR . . . 8
- 3.3 Implementation in TPDE . . . 8

4 Scopes and Types **11**
- 4.1 Representation in DWARF . . . 11
  - 4.1.1 Tree of Debugging Information Entries . . . 11
  - 4.1.2 Attributes and Tags . . . 12
  - 4.1.3 Functions and Inlining . . . 14
  - 4.1.4 Encoding . . . 16
- 4.2 Encoding in LLVM IR . . . 18
  - 4.2.1 Scopes . . . 18
  - 4.2.2 Types . . . 18
  - 4.2.3 Parameters and Variables . . . 19
  - 4.2.4 Inlining . . . 19
- 4.3 Implementation in TPDE . . . 19
  - 4.3.1 Tree Data Structure . . . 20
  - 4.3.2 Tree Construction . . . 21
  - 4.3.3 Ranges . . . 23
  - 4.3.4 Tree Serialization . . . 23
- 4.4 Limitations . . . 25

5 Variable Locations **27**
- 5.1 Representation in DWARF . . . 27
  - 5.1.1 Expressions . . . 27
  - 5.1.2 Location Lists . . . 29
- 5.2 Encoding in LLVM IR . . . 30
  - 5.2.1 Debug Records . . . 31
  - 5.2.2 Expressions . . . 32

5.3 Implementation in TPDE 33
5.3.1 Location List Data Structure 33
5.3.2 Location List Construction 34
5.3.3 Propagation of Debug Records 35

6 Evaluation **39**
6.1 Local Variable Coverage 39
6.2 Compilation Time 42
6.2.1 Setup 42
6.2.2 Results and Discussion 42
6.3 Section Sizes 45

7 Summary **47**

Acronyms **49**

Bibliography **51**

1 Introduction

Typical tasks of a programmer include writing code, building it and testing it. Together, these tasks form a cycle repeated numerous times during work days. Time is used productively when writing and testing code, whereas waiting for a build to finish is a primarily passive activity, which can furthermore distract the programmer if it takes too long. However, especially when working on larger projects, compilation of source code can require a considerable amount of time.

One cause of high build times is the compilation process commonly employed in compilers like GCC [3] and Clang [8], which run passes in which they successively iterate over and refine intermediate representations of the program. This architecture is also found in the respective target-specific compiler back-ends: Even if optimizations are disabled, the number of back-end passes is typically in the order of 70 passes[1], of which each contributes to compilation time.

This is exacerbated by the fact that during development, a programmer usually instructs the compiler to emit *debugging information*. In the case of an error needing further investigation, debugging information aids in finding the cause, because it maps instructions to source lines, provides information about scopes and types of program entities and describes the locations of variables in registers and memory. However, maintaining these pieces of information throughout the execution of numerous passes slows these down.

One approach to reducing compilation times is TPDE [15]: It is a compiler back-end framework for the x86-64 architecture that reduces the number of passes to only 3. It reads LLVM IR [10] and can thus be employed as an alternative LLVM [13] back-end, especially as part of the compiler Clang.

TPDE first places global values in the buffers for the respective object file sections, so that it can use their addresses during function compilation. Per function, it then runs following passes: In the first pass, TPDE iterates through LLVM IR to fill lookup structures and to additionally replace unsupported constructs. Next, it performs loop and liveness analysis. The results of the analyses are used in the last pass, in which TPDE performs code generation. This pass combines instruction selection, register allocation and machine code emission and additionally produces unwinding information for exception handling. After function compilation, TPDE combines the sections to form an object file that can be written to the filesystem or executed from memory.

At optimization level `O0`, TPDE compiles LLVM IR about 10–20 times faster than LLVM's back-end, while the runtime performance of generated code is comparable [15]. However, TPDE does not support the generation of debugging information, limiting its usability in development.

[1]We measured 73 passes for LLVM 19.1.0-rc2 using `llc -O0 -debug-pass=Structure`.

In this thesis, we extend TPDE with functionality to emit debugging information in the DWARF [2] format, which is the default format on Unix-like operating systems such as GNU/Linux. We focus on the C and C++ languages for source files and we prioritize a low slowdown of the back-end over space-efficient encoding. Our implementation provides debuggers with line number information, data about scopes and types and with locations of local variables. Due to the limited scope of this thesis, our implementation is currently not capable of encoding locations of global variables. Furthermore, we do not extend TPDE's existing functionality of generating stack unwinding information.

Key results include that our additions to TPDE slow down compilation by an average of 45% when compiling the SPECspeed 2017 Integer benchmarks. Thus, the speedup of TPDE relative to LLVM is only marginally lowered when compiling with debugging information. While compilation with TPDE increases the size of debugging information by 22% on average, we cannot name definitive results for variable coverage.

Outline

This thesis is structured as follows: In Chapter 2 we will name different components of debugging information and point out concepts of LLVM IR and DWARF that are common to all parts. The next three chapters will be dedicated to the individual components: In Chapter 3, we will focus on line number information, in Chapter 4 on the graph of program entities and in Chapter 5, we will concentrate on variable locations. In Chapter 6 we will evaluate our implementation with respect to the impact of our modifications on compilation time and we will also consider the variable coverage and section sizes of produced debugging information. Finally, we will summarize the results of this thesis in Chapter 7.

Contributions

Key contributions of this thesis include:

- A non-exhaustive summary of information required by current debuggers.
- An implementation of a DWARF producer embedded into a three-pass LLVM IR compiler back-end.

2 Technical Background

In this section we provide an overview of the parts debugging information is typically composed of. Later, we name concepts of DWARF and LLVM that are independent of the different parts we discuss in chapters 3–5.

2.1 Debugging Information

Debuggging information comprises various mappings between entities in a program's source and the corresponding elements in the binary.

One such mapping is *line number information*, which translates instruction addresses to the locations of statements in the source that produced the instructions. Line number information is essential for interactive debuggers, which employ it for features like stepping through program statements and setting breakpoints at source lines.

Another mapping typically found in debugging information links instructions to source-level *scopes*. This is needed, for example, when a debugger prints details about the context of the currently executed statement. In languages like C and C++, scopes are furthermore closely connected to *types*. Information about these is required when operating on typed entities such as constants and variables.

Since the values of the latter are not fixed, a debugger requires descriptions where to find them. Because of program optimizations, the *variable locations* may themselves not be constant during execution. Therefore, their descriptions in debugging information depend on the instruction currently executed by the program.

Especially in the event of a program crash, a user might want to inspect the local variables of the function in which the crash occured and those of its callers. For that purpose, debugging information can provide *call frame information* that helps to find the stack frames of calling functions.

2.2 DWARF

DWARF [2] is the predominant debugging information format in Linux and Unix environments. Typical producers of DWARF are compilers, though we are also aware of non-compiler producers, such as ghidra2dwarf[1], which inserts debugging information obtained by reverse-engineering into an executable. Consumers primarily include interactive debuggers, in particular the two debuggers GDB [4] and LLDB [12].

DWARF relies on various features of the containing object file format. In particular, it requires dedicated sections in which it encodes different parts of debugging information. References between these sections furthermore require relocation during linking. DWARF

[1] `https://github.com/cesena/ghidra2dwarf`, accessed 2024-08-27

sections from multiple compilation units are concatenated by the linker, which can lead to duplicate information.

DWARF is a binary format, which, for the purpose of shorter section sizes, makes extensive use of variable length encodings, especially of Little-Endian Base 128 (LEB128). LEB128 encodes integers by splitting them into parts of 7 bits and storing these parts in separate bytes. The remaining bit of each byte is used to mark the end of an LEB128-encoded integer. Another common pattern aiming to reduce encoded size is that DWARF offers multiple encodings for the same value. A DWARF consumer can choose the most compact encoding for a given set of values.

The DWARF specification [2] is currently available in version 5. To enable consumers which do not support this version to detect and skip DWARF5 debugging information, DWARF includes version numbers in object file sections in which the encoding deviates from the previous versions. Thus, different versions of DWARF can coexist in one object file.

2.3 LLVM

LLVM[2] [13] is a modular compiler infrastructure built around its intermediate representation LLVM IR [10]. It comprises multiple compiler frontends generating LLVM IR—most notably the C and C++ compiler Clang [8]—features a collection of analysis and optimization passes as well as back-ends for different targets such as x86-64 and AArch64.

Based on Static Single Assignment (SSA), every function in LLVM IR is represented as a control flow graph whose nodes are basic blocks. Each basic block is a sequence of instructions that can only be executed as a whole as long as no exceptions occur.

When in memory, program entities are encoded as structures that are linked together via pointers. In particular, LLVM IR makes heavy use of linked lists. One example of such a linked list is the sequence of independently allocated instructions stored for each basic block. These pointer-heavy structures allow LLVM IR passes to perform insertions and deletions with low asymptotic complexity at the cost of degraded cache locality.

In addition to its *in-memory* representation, LLVM IR can also be serialized to *bitcode* that allows caching and passing of intermediate representation between tools. Furthermore, LLVM IR also has a human readable *textual* form that is useful for debugging. This is the form chosen for examples in this thesis. As all three forms of LLVM IR are equivalent, an example shown in the *textual* form does not mean that LLVM IR is actually serialized.

Debugging information is optional in LLVM IR. When present, entries in the intermediate representation contain references to specialized metadata nodes and strings. Thus, only space for one additional pointer is required in LLVM IR structures, which saves space in the case that no debugging information is provided. In the textual form, references to metadata nodes can be recognized by a leading exclamation mark, such as in `!18` or in `!namedReferenceToNode`. The specific locations in LLVM IR where debugging information is referenced are discussed in the respective chapters of this thesis.

[2]All results of this thesis related to LLVM and its subprojects Clang and LLDB were obtained using version 19.1.0-rc2, Url: `https://github.com/llvm/llvm-project/tree/llvmorg-19.1.0-rc2`, accessed 2024-08-27

3 Line Table

The line table provides a mapping from instruction addresses to the file locations of program source entities the instructions originate from. Despite being called *line table*, this mapping can also provide additional per-instruction information such as the used instruction set and aid the consumer in reconstructing the program's control flow graph.

3.1 Representation in DWARF

The line table is stored in the `.debug_line` section and is conceptually standalone. It is a compressed encoding of a matrix that contains one row for each instruction.

3.1.1 Fields

The fields in each row can be grouped as follows: [2]

Reference to the Instruction The field `address` specifies the location of the instruction in the program's address space. The field `op_index` contains the index of the referred-to operation in the case of an instruction that encodes multiple operations, like an VLIW instruction. `op_index` is not relevant on x86-64 and therefore always set to 0.

Source Location The integer fields `file`, `line` and `column` provide the source code location of the program statement that the instruction belongs to. `file` is an index into an array containing per-file information. This array is also stored in the `.debug_line` section. The DWARF producer can choose which per-file information it adds. When no extensions are used, possible options are the filename, the directory, the time of last modification, the file's size and an MD5 checksum. Directory information is encoded as an index into a directory array containing directory paths. Filenames and directory paths are stored in a separate string table section, `.debug_line_str`.

Instruction Set On architectures that support switching between multiple instruction sets, the integer field `isa` provides information about the currently used instruction set. `isa` is always set to 0 on x86-64.

Breakpoint Hints Via the three boolean flags `prologue_end`, `epilogue_begin` and `is_stmt` a compiler can hint a debugger where to place breakpoints: An instruction annotated with `prologue_end` is the first instruction after a function prologue. When a user requests to halt the program on entry of a function, inserting a breakpoint at an instruction marked with `prologue_end` yields better debugging experience than breaking at

```c
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}
```

Listing 3.1: *Simple example source file: sum.c*

	address	line	column	file	prologue_end	epilogue_begin	is_stmt
`sum:`							
`push rbp`	0x1130	4	0	0	0	0	1
`mov rbp,rsp`	0x1131	4	0	0	0	0	1
`mov DWORD PTR [rbp-0x8], edi`	0x1134	4	0	0	0	0	1
`mov DWORD PTR [rbp-0x4], esi`	0x1137	4	0	0	0	0	1
`lea eax,[rdi+rsi*1]`	0x113a	5	14	0	1	0	1
`pop rbp`	0x113d	5	5	0	0	1	0
`ret`	0x113e	5	5	0	0	0	0

Figure 3.1: *x86-64 assembly and corresponding uncompressed line table generated for sum.c using* `clang -g -O0`*. Only fields relevant for x86-64 debuggers are shown. Addresses are not relocated.*

the entry label, as the stack frame is already set up and can be inspected. Similarly, when a user requests a breakpoint before function exit, the debugger can use the `epilogue_begin` flag to find suitable breakpoint locations before the stack frame is torn down. `is_stmt` denotes that an instruction is a recommended breakpoint location and therefore helps the compiler to select one of multiple instructions that map to the same source location.

The degree of adoption of these hints varies between different producers and consumers. For example, the LLVM back-end produces these hints, while GCC 13.1.0 does not emit them at all. Also, GDB currently[1] does not consider `prologue_end` to skip function prologues when single-stepping, while it skips prologues based on `prologue_end` when setting a breakpoint on a function entry. LLDB, however, follows these hints in both cases.

Control Flow Information The flag `basic_block` indicates the beginning of a basic block and the field `discriminator` can be filled with arbitrarily chosen values that link instructions to their basic blocks. These fields help profilers to reconstruct the control flow of programs. Debuggers usually ignore both fields.

3.1.2 Compression

Figure 3.1 shows an uncompressed line table for the source file in Listing 3.1. Whereas the full uncompressed matrix describes the semantics of the DWARF line table well, this

[1]Commit: 46b6ba96, `https://sourceware.org/git/?p=binutils-gdb.git;a=commit;h=46b6ba96`, accessed 2024-08-29

address	line	column	file	prologue_end	epilogue_begin	is_stmt
0x1130	4	0	0	0	0	1
0x113a	5	14	0	1	0	1
0x113d	5	5	0	0	1	0
0x113f	-	-	-	-	-	-

Figure 3.2: *Line table from Figure 3.1 without redundant rows. The last row is a sentinel row.*

```
DW_LNS_set_file 0
DW_LNE_set_address 0x1130
special: address += 0, line += 3
DW_LNS_set_column 14
DW_LNS_set_prologue_end
special: address += 10, line += 1
DW_LNS_set_column 5
DW_LNS_negate_stmt
DW_LNS_set_epilogue_begin
special: address += 3, line += 0
DW_LNS_advance_pc: address += 2
DW_LNE_end_sequence
```

Listing 3.2: *Line number program generating Figure 3.2*

form of the line table is not used for storage as its size would be a multiple of the code section size.

Instead, for each contiguous sequence of instructions, the corresponding rows of the matrix are grouped together and sorted by ascending `address` and `op_index` values. Then, all rows that only differ from the previous row in that they refer to a different instruction, are dropped if none of `basic_block`, `prologue_end` or `epilogue_begin` are set. As rows now refer to instruction ranges, the end of each continuous instruction sequence has to be marked with a sentinel row.

While GDB and LLDB use an in-memory representation similar to this *reduced matrix*, the line table still has low entropy and therefore undergoes additional compression when serialized to disk. For this purpose, the DWARF standard specifies an encoding for *line number programs* that produce the reduced matrix when executed by a state machine that has a register for each of the fields. Instead of the matrix, the line number program is stored in the `.debug_line` section.

Line number programs consist of operations that can modify register values, flush the current state of the registers into a new row or mark the end of an instruction sequence. The lengths of opcodes vary between operations. Operations that are expected to appear frequently have a one-byte encoding. Additionally, a major part of the one-byte opcode space is assigned to *special opcodes* that increment the instruction reference, modify the line number and emit a new row in one operation. The range of possible values for the increments and offsets can be adjusted by the DWARF producer, which allows a tradeoff between being able to efficiently encode larger address increments or larger line offsets.

Figure 3.2 contains the reduced matrix equivalent to Figure 3.1. As the rows in the uncompressed line table are already sorted by `address` and as there is only one contiguous sequence of instructions, no reordering is needed in this example.

In the line number program in Listing 3.2, `address` is only set at the beginning of the sequence and then incremented between each row using special opcodes and `DW_LNS_advance_pc`. That is because increments not only have a more efficient encoding but also they do not require a relocation in contrast to `DW_LNS_set_address`. It can can also be seen that the flags `prologue_end` and `epilogue_begin` are automatically reset after each row.

```
define i32 @sum(i32 %a, i32 %b) !dbg !2 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, ptr %a.addr, align 4
  store i32 %b, ptr %b.addr, align 4
  %0 = load i32, ptr %a.addr, align 4, !dbg !3
  %1 = load i32, ptr %b.addr, align 4, !dbg !4
  %add = add nsw i32 %0, %1, !dbg !5
  ret i32 %add, !dbg !6
}
```

(a) *Function* `sum`*. Attributes and debug records have been removed for simplicity.*

```
!1 = !DIFile(filename: "sum.c", directory: "/tmp",
     checksumkind: CSK_MD5, checksum: "363
     a6746a516c2b5b3a1da30b11a5285")
!2 = distinct !DISubprogram(name: "sum", scope: !1,
      file: !1, line: 3, scopeLine: 4, [...])
!3 = !DILocation(line: 5, column: 12, scope: !2)
!4 = !DILocation(line: 5, column: 16, scope: !2)
!5 = !DILocation(line: 5, column: 14, scope: !2)
!6 = !DILocation(line: 5, column: 5, scope: !2)
```

(b) *Corresponding metadata nodes carrying source location information. Additional fields in* `DISubprogram` *are omitted.*

Figure 3.4: *LLVM IR for sum.c (Listing 3.1) generated by* `clang -g -O0`

DWARF defines further operations that are not mentioned in this section. We refer the reader to the standard [2] for further information.

3.2 Encoding in LLVM IR

Instructions that have an associated source code location contain a reference to a `DILocation` metadata node. In the assembly form, these references can be recognized by their `!dbg` prefix, as shown in Figure 3.4a. Each `DILocation` node provides information for the in-file location via the fields `line` and `column`.

The file itself is given indirectly via a reference to the containing scope of the program statement that generated the instruction. The type of the scope node can be any one of the `DILocalScope` subclasses, of which `DISubprogram` and `DILexicalBlock` are the most common. The former describes a function or method and the latter provides information about a block within a function or method. Like all metadata nodes representing scopes, these nodes contain a `file` field that references a `DIFile` node, which provides the filename and directory and optionally also a checksum.

Figure 3.4b shows that the scope nodes also feature a *line* field. This field is not relevant for the line table as it describes, where the function was defined, which is not necessarily a location of a statement that generated an instruction. However, the `scopeLine` value provided by `DISubprogram` is an appropriate choice for a line number to assign to the function prologue, because it marks the beginning of the function's scope.

3.3 Implementation in TPDE

In TPDE, we generate and serialize the line number program directly from LLVM metadata and from instruction addresses without any previous transformations. The line table is produced together with the machine code in the last compiler pass. We only generate values for the fields that are relevant to x86-64 debuggers and we do not generate `is_stmt` hints since these are not directly obtainable from LLVM IR. Our approach exploits following two properties of TPDE:

First, TPDE sequentially emits machine code into a memory buffer and never reorders or moves already generated instructions. Although there are situations, in which TPDE needs to update constant operands of already written instructions or replace instructions in function prologues and epilogues, this does not affect instruction addresses relevant for the line table.

Second, since source location information is provided by LLVM instructions, that information cannot change in between two machine code instructions that are result of the same LLVM instruction.

This enables us to hypothetically generate a reduced matrix as defined in 3.1 by recording a row each time before compiling an LLVM instruction. The `address` of the next machine instruction is obtained from the write pointer offset in the code buffer and the source location fields are read from the next LLVM instruction.

In order to directly generate the line number program, we do not record a row each time, but instead simulate the state machine's registers. Whenever a new row for the reduced matrix would be emitted, we assign its values to the simulated registers and check if any of these except `address` have changed. Only in the case of changed values we emit opcodes that update the registers of the consumer's state machine and produce a new row.

To provide line number information for prologues and epilogues, we also emit a row before each prologue and set the `prologue_end` and `epilogue_begin` flags after compiling prologues and before compiling epilogues respectively.

Per-File Information TPDE transfers the information provided in `DIFile` nodes into the per-file information array found in the `.debug_line` section. Because LLVM does not provide file sizes and access times, these attributes are also not provided by TPDE. Furthermore, TPDE does not copy MD5 checksums, even when they are provided by LLVM. That is, because these checksums are optional and not reliably provided by LLVM IR producers, whereas DWARF only offers to to either provide checksums for all files or for none.

Although duplicate file entries are not forbidden by the DWARF standard, the increased section size would be impractical, if per-file information were duplicated for each row. TPDE must therefore remember which `DIFile` nodes it has already added to the array.

For this purpose, we added a hash map that translates the `DIFile` pointer to its corresponding array index if it was already seen. To accelerate the map lookup in the common case that two consecutive instructions refer to the same file, TPDE also remembers the pointer used for the most recent lookup and the lookup result. We use a similar pattern with the metadata string pointer as key to prevent duplicate entries in the directory name array.

4 Scopes and Types

Program entities relevant for debuggers are connected to other entities because of relations defined by the source language semantics. This results in a graph that the debugging information format has to define an encoding for.

In the C and C++ programming languages, two ubiquitous relations are the scope hierarchy and the connections between types. Most scopes have no dedicated definition. Instead, these scopes are implicitly provided by program entities such as functions and classes. Scopes can be nested and thus form a tree that is rooted at global scope.

In contrast, the relation between types is not a tree. This is due to the fact that there are instances in which types are derived from multiple base or member types. For example, C++ allows multiple inheritance. Also, structures and unions can have multiple members of different types. Because these member types can in turn be derived from or be pointers or references to the structure or union type, the type relation can also contain cycles.

4.1 Representation in DWARF

DWARF stores the graph of program entities and their attributes in the `.debug_info` section. All other debugging information except acceleration tables is referenced from `.debug_info`. The `.debug_info` section thus acts as the entry for a debugger when parsing DWARF. [2]

4.1.1 Tree of Debugging Information Entries

The contents of the `.debug_info` section are organized as an ordered tree of *Debugging Information Entries (DIEs)*. Each DIE describes a program entity and its attributes. The tree models ownership: A parent DIE owns its child DIEs. This ownership relation contains the scope relation, which means that any DIE describing a scope owns all DIEs which describe program entities contained in that scope. DWARF does not impose any restriction on the order of these child DIEs.

Furthermore, if the parent DIE describes an entity composed of parts, then these parts are described by child DIEs whose order is the same as that of the parts. One example of such a composed program entity are structure types: The member types of a structure are provided by dedicated child DIEs, as seen in Figure 4.1, in which the member types of `struct list_node` reference DIEs that encode the types `int` and `struct list_node*`.

Thus, the ownership relation additionally includes parts of the type relation. The remaining connections between types as well as the other relations within the graph of program entities are provided by references that are stored as attributes in the DIEs.

```c
#include <stddef.h>

struct list_node {
    int value;
    struct list_node* next;
};

int count_nodes(struct list_node* list) {
    int result = 0;
    while(list != NULL) {
        list = list->next;
        result++;
    }
    return result;
}
```

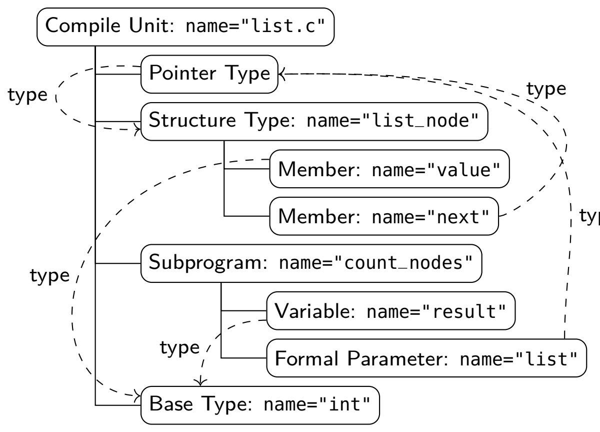

Compile Unit: name="list.c"
Pointer Type
Structure Type: name="list_node"
Member: name="value"
Member: name="next"
Subprogram: name="count_nodes"
Variable: name="result"
Formal Parameter: name="list"
Base Type: name="int"
type
type
type
type
type

Figure 4.1: *Exemplary source file and corresponding tree of DIEs as generated by TPDE. References are visualized as dotted lines. Attributes except name and type are omitted.*

The program in Figure 4.1 additionally features a cycle in the type relation: The structure type `list_node` contains a pointer to itself. It also shows that DIEs encoding entities without a scope are owned by the root node, which describes the compile unit.

4.1.2 Attributes and Tags

Apart from its child DIEs, each DIE comprises a *tag* and a set of *attributes* with their corresponding values. The tag is an integer that defines the kind of the described program entity and thus also the set of applicable attributes and their meanings. Attributes are provided as key-value pairs. The keys are integers and possible classes for the values include, among others, integers, boolean flags, strings and references to other DIEs or to contents of other sections.

Allowed values for tags and attribute keys are defined by the DWARF standard. Their symbolic constants start with `DW_TAG_` and `DW_AT_` respectively. Not all of these are relevant to C and C++ compilers since the DWARF standard also covers other languages to varying degrees. Vendors can define additional tags and attributes.

DWARF does not specify the meaning for every possible combination of tags and attributes. Combinations not described by the standard may be emitted by producers, but have an increased likelihood of providing no additional information to the consumer, which is required to ignore attributes it does not understand. Below, we provide an overview of a selection of attributes that have a defined meaning across multiple different tags.

Names The names of program entities can be provided via `DW_AT_name`. The absence of names can be encoded either by providing an empty string or by not providing the attribute `DW_AT_name` at all. In cases in which the name provided by the source code and the name as visible to the linker are not the same, the source-level name is that encoded by `DW_AT_name` and the linkage name is provided via an additional `DW_AT_linkage_name` attribute. This is for example relevant for C++, which relies on mangled linkage names to distinguish entites that have the same name but differ in types or scopes.

Declaration Coordinates By providing one or multiple of the attributes `DW_AT_decl_file`, `DW_AT_decl_line` and `DW_AT_decl_column`, a producer can describe the location of an entity's declaration in the source code. The attributes have the same meanings as the fields `file`, `line` and `column` of the line table respectively. In particular, `DW_AT_decl_file` always encodes an index into the file array in the `.debug_line` section, although Section 2.14 of the DWARF standard requests that the value zero for `DW_AT_decl_file` encode the absence of file information. However, this inconsistency with the definition of the `file` field was erroneously copied from the fourth DWARF version, where value zero in the line table's `file` field also represents missing file information. In the upcoming sixth DWARF version, this issue will be fixed [1]. Clang relies on debuggers to interprete zero as a valid index into the file array, while GCC 13.1.0 avoids the issue by duplicating the first entry in the file array so that it can use the index one to refer to that file.

Program Counter Ranges When describing entities which produced machine code, such as compilation units, functions and and scopes within functions, DWARF producers can append information about the range of the instruction's addresses to the DIE. Contiguous ranges can be provided using the `DW_AT_low_pc` and `DW_AT_high_pc` attributes. These attributes encode the addresses of the first instruction within and the first instruction past the range respectively. Discontiguous ranges require the use of the `DW_AT_ranges` attribute, whose value is a reference to a *range list* contained in the `.debug_rnglists` section. Each range list describes multiple contiguous ranges that together form a discontiguous range.

Types DIEs describing typed entities such as variables, formal function parameters, constants and members of composed types have a `DW_AT_type` attribute which references another DIE describing the type. Furthermore, the `DW_AT_type` attribute is also used in DIEs that describe a derived type to provide a reference to the base type. Possible derived types include, among others, pointer and reference types, const-qualified types, and arrays types.

In Figure 4.1, the tree's root is a DIE describing the compilation unit. This DIE has the tag `DW_TAG_compile_unit`. A DWARF producer can also generate contributions to the `.debug_line` section whose roots contain different tags. This is however only possible if the producer employs mechanisms to reuse types information or descriptions of imported modules or mechanisms to store debugging information outside of the object file or executable. As we do not implement any of these mechanisms in TPDE and do not instruct other compilers to enable these features, we assume that every root DIE is a `DW_TAG_compile_unit` DIE in the remainder of this thesis.

The compilation unit DIE serves multiple purposes: Its attributes provide information about the compilation, such as the name and the programming language of the source

[1] See `https://dwarfstd.org/issues/210713.1.html`, Accessed: 2024-08-13

file, the directory the compile command was executed in, and the name of the producer. Additionally they provide data needed to decode other attributes of the compile unit DIE and its child DIEs. That information can include the encoding of strings and relocated offsets to the contributions to other sections that contain referenced content. Furthermore, the compile unit DIE acts as DIE that describes global scope and it also contains DIEs of entities that neither a part of another program entity have a scope.

As shown in Figure 4.1, one example of such DIEs are DIEs describing basic types. These DIEs have the tag `DW_TAG_base_type` and contain attributes describing the size and encoding of the basic type, where encoding refers to the particular kind of basic type. Available encodings include signed and unsigned integers, floating point numbers and characters of various character sets.

Other DIEs that encode type information include those with the tags `DW_AT_class_type`, `DW_AT_structure_type` and `DW_AT_union_type`, which are used to describe the C and C++ composed type constructs `class`, `struct` and `union` respectively. In addition to their child DIEs that provide the data member types, they may also contain DIEs representing member functions, static variables and inherited types. The order of the member and inheritance child DIEs relative to each other has to match that of their declarations in the source file. The other child DIEs can however be ordered freely, which also includes positions between member and inheritane DIEs.

4.1.3 Functions and Inlining

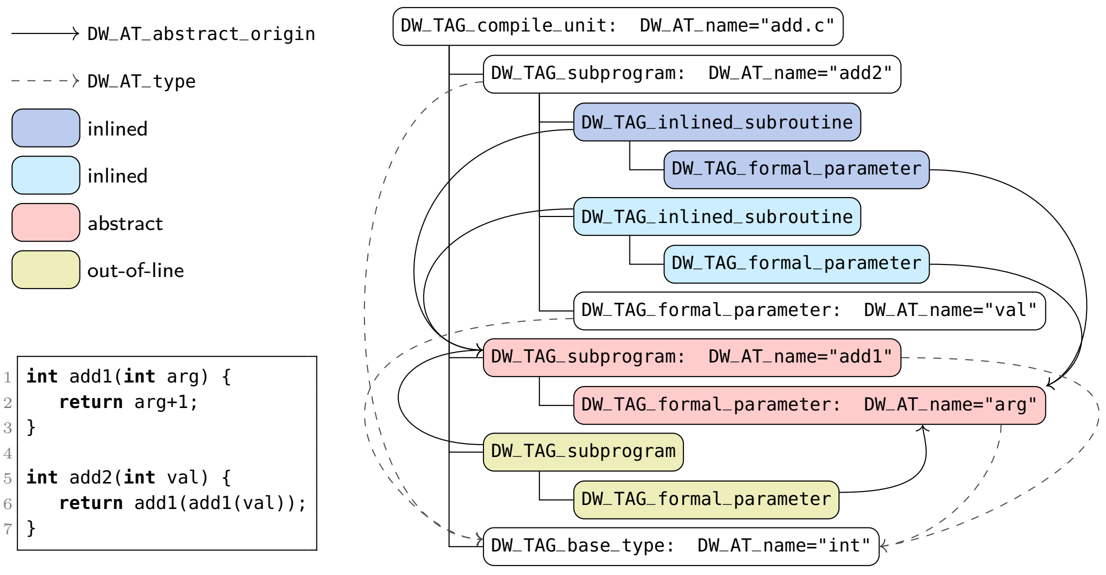
DIEs with the tag `DW_TAG_subprogram` describe functions and member functions. Since functions contribute to the generated machine code, a program counter range can be specified using the attributes described in Section 4.1.2.

Although DWARF provides means to encode function types via `DW_TAG_subroutine_type` DIEs—these are used for example for function pointers—the `DW_AT_type` attribute in subprogram DIEs only refers to the return type of the described function. The absence of a return value, which corresponds to return type `void` in C and C++, is represented by the absence of the `DW_AT_type` attribute.

The argument types, including `this` for C++ member functions, are indirectly referenced via child DIEs of tag `DW_TAG_formal_parameter`. Each subprogram DIE contains one of these DIEs for each formal parameter, which also provides the name. Similar to the members of composed types, only the order of the formal parameter DIEs relative to each other has to match the order of the function's formal parameters.

Subscopes of functions are encoded as `DW_TAG_lexical_block` DIEs. All DIEs representing a function's scopes, including the subroutine DIE itself, can contain `DW_TAG_variable` DIEs, which then describe local variables and contain a set of attributes similar to that of formal parameters.

In addition to the already mentioned tags and attributes related to functions, information about inlined function calls can be provided. If it is omitted, and a debugger pauses program execution at an instruction that belongs to an inlined function call, the debugger erroneously assumes that the variables of the caller are in scope, even if it displays the correct location of the program counter in terms of source code. This leads do dissatisfying debugging experience.


DW_AT_abstract_origin
DW_AT_type
inlined
inlined
abstract
out-of-line
DW_TAG_compile_unit: DW_AT_name="add.c"
DW_TAG_subprogram: DW_AT_name="add2"
DW_TAG_inlined_subroutine
DW_TAG_formal_parameter
DW_TAG_inlined_subroutine
DW_TAG_formal_parameter
DW_TAG_formal_parameter: DW_AT_name="val"
DW_TAG_subprogram: DW_AT_name="add1"
DW_TAG_formal_parameter: DW_AT_name="arg"
DW_TAG_subprogram
DW_TAG_formal_parameter
DW_TAG_base_type: DW_AT_name="int"

```c
int add1(int arg) {
   return arg+1;
}

int add2(int val) {
   return add1(add1(val));
}
```

Figure 4.2: *A tree of DIEs that contains abstract and concrete instance trees. Each instance tree is colored differently. The tree was generated from the source on the left using* `clang -g -O2` *with the TPDE back-end. All attributes except* `DW_AT_name`*,* `DW_AT_type` *and* `DW_AT_abstract_origin` *are omitted.*

Inlined function calls in DWARF are encoded as *concrete inlined instance trees*. These are subtrees starting at DIEs with the tag `DW_TAG_inlined_subroutine` that are owned by the DIEs that describe the scopes in which the calls occur. Via the `DW_AT_abstract_origin` attribute, each inlined subroutine DIE points to a `DW_TAG_subroutine` node, which is marked as the root node of an *abstract instance tree*. An abstract instance tree differs from regular subtrees of subroutine DIEs in that its DIEs do not feature attributes that are specific to any compiled instance of the subroutine. That means that attributes such as `DW_AT_name` are provided in abstract instance trees, while attributes such as `DW_AT_ranges` are not. The structure of a concrete inlined instance tree mirrors that of its abstract counterpart. It can, however, contain additional DIEs if there is information which is only relevant for this particular inline expansion of the function, and it can omit DIEs if they would not provide additional information relative to their abstract counterpart. Each node in a concrete inlined instance tree which has an equivalent in the abstract instance tree references it as its abstract origin and omits attributes already defined in the abstract DIE.

When non-inlined code is generated for a function that is also inlined in another function, then an additional *concrete out-of-line instance tree* describes this non-inlined instance of the function. The only differences between concrete out-of-line instance trees and concrete inlined instance trees are that the former's root node has the tag `DW_TAG_subprogram` and is not owned by a DIE representing a local scope.

Figure 4.2 shows an example of inlining: The function `add1` is inlined twice in `add2`. Thus, the `DW_TAG_subprogram` DIE for `add2` contains two concrete inlined instance trees, whose DIEs share their counterparts in the abstract instance tree. Additionally, there is a concrete out-of-line instance tree that in this example—that order is however not required—succeeds the abstract instance tree. The DIEs in the concrete instance trees do not provide the `DW_AT_type` attribute as it is indirectly obtainable via `DW_AT_abstract_origin`.

4.1.4 Encoding

Since different DIEs with the same tag often share their set of attribute keys and datatypes, only the attribute values are encoded per-DIE. The keys and datatypes are stored out-of-line in in the `.debug_abbrev` section.

`.debug_abbrev` section Each contribution to `.debug_abbrev` contains a sequence of *abbreviations*, each starting with a unique LEB128-encoded *abbreviation code*. The codes can be any positive integers and the standard does not require the abbreviations be sorted by their codes. Following the abbreviation code, each abbreviation stores the tag that is common to the DIEs that reference this abbreviation. The next byte is a boolean flag indicating whether the DIEs can have children. The remaining parts of the abbreviation are pairs of LEB128-encoded attribute keys and *forms*.

Forms are integers that define both the class and the encoding of a value. For example, values of the form `DW_FORM_strp` are strings that are provided as an offset into the `.debug_str` section, whereas `DW_FORM_str` is used for strings that are provided directly in the DIE. Another example are the two forms for boolean flags: DWARF consumers consider boolean attributes that are not included in DIEs as false. Therefore the value of a flag can be encoded as the presence of the attribute, in which case the form `DW_FORM_flag_present` must be used. This form describes a boolean flag that is always true and whose value encoding does not occupy any space in the DIE. Alternatively, flags can also be provided as one-byte values of the form `DW_AT_flag`, in which case the value 0 encodes false. The set of forms is fixed by DWARF. No further forms can be added by compatible extensions to ensure that consumers can always skip attributes they do not understand.

The sequence of attribute keys and forms is terminated by a sentinel pair which consists of two zeros. Analogously, the sequence of abbreviations is terminated by a zero byte. The order of attribute keys has no semantic meaning in DWARF.

`.debug_info` section Every contribution to the `.debug_info` section starts with a header, whose fields include, among others, the length of the contribution and the DWARF version. The header is followed by the encoding of the tree, which is serialized in pre-order. Thus, the first encoded DIE is the root DIE followed by its children.

Each DIE starts with a LEB128-encoded abbreviation code linking it to the DIE's abbreviation. Next, the encodings for the attribute values are appended in the order of their attribute keys in the abbreviation. Table 4.1 shows a possible encoding of a `DW_TAG_base_type` DIE which represents a four-byte signed integer.

If the abbreviation of a DIE states that the DIE can contain children, then its encoding is followed by the encodings of its children, which appear in the same order as in the tree. The sequence of child DIE encodings is terminated by a zero byte. This zero byte can also be interpreted as a DIE with abbreviation code zero, which is reserved for this purpose.

Table 4.1: *Encoding of a `DW_TAG_base_type` DIE showing the bytes of the abbreviation on the left and the bytes stored in the `.debug_info` section on the right in hexadecimal format. We note that in this example LEB128 encoding always results in a single byte, because the values of the abbreviation code, the tag and all attribute keys and forms are less than 128. The name string is represented as an offset into the `.debug_str` section and thus requires a relocation. In the example, the offset is a four-byte little endian value.*

`.debug_abbrev`	Interpretation			`.debug_info`
`04`	Abbreviation Code: 4			`04`
`24`	Tag: `base_type`			
`00`	DIE has no children			
	Attribute	Form	Value	
`03 0e`	`name`	`strp`	String at `.debug_str+7`	`07 00 00 00`
`3e 0b`	`encoding`	`data1`	`DW_ATE_signed`	`05`
`0b 0b`	`byte_size`	`data1`	4	`04`

Because the number of encoded bytes for each attribute value depends on the form, a consumer has to consider the abbreviation to find the boundaries between attribute values. Furthermore, the length of an attribute value can also depend on the value itself in the case of variable length data. Thus, a consumer has to process all preceding attributes and DIEs to read a specific attribute of a DIE. A producer can optionally provide a reference to the next non-descendant DIE via the `DW_AT_sibling` attribute, which offers the consumer a means to skip subtrees when reading. Because of the additional space required, this attribute can also degrade debugging experience. LLVM thus does not emit it at all [19].[2]

References to other DIEs which are part of the same contribution to `.debug_info` are encoded as offsets relative to the beginning of the contribution's header. References to contents of other sections are either indices into arrays or offsets into the corresponding sections. Offsets can be absolute, in which case they require relocation, or relative to an absolute offset provided by an attribute of the compile unit DIE. Because of the saved relocations, relative offsets are preferred over absolute offsets where feasible.

`.debug_rnglists` section Producers of DWARF version 5 debugging information can choose between multiple encodings for the ranges in `.debug_rnglists`. The used encoding is specified individually for each entry of a range list in the entry's fist byte. DWARF provides encodings, which are based on address-sized integers that require relocation or on indices into an `.debug_addr` section. This section contains an array of addresses that also need to be relocated but allow reusing, which can save relocations. Other encodings utilize offsets that are relative to a base address, which is either the `DW_AT_low_pc` attribute of the compile unit or a value provided in a preceeding entry of range list. Each range list entry that is not used to mark the end of a list or to provide a base address contains either encodings for the start and end or for the start and length of the range. The ranges in the range list do not have to be sorted, but they may not intersect.

[2]Also mentioned in `https://reviews.llvm.org/D43439`, accessed 2024-08-17

4.2 Encoding in LLVM IR

All debugging information metadata nodes which correspond to a DIE in the `.debug_info` section are instances of the `DINode` class. There are various subclasses of `DINode`. In this section, we will cover those that are relevant to the scope of this thesis. [10]

4.2.1 Scopes

All debugging information metadata nodes describing scoped program entities feature a `scope` field. This field points to a node whose type is a subtype of `DIScope`. All `DIScope` nodes describe scoped entities themselves and therefore they also feature the `scope` field.

`DIScope` nodes of types `DIFile` and `DICompileUnit` respresent the global scope. Additionally, if `scope` is a null reference, this also indicates that the described entity has global scope. Thus, the scope relation in LLVM metadata nodes forms a forest instead of a tree.

4.2.2 Types

LLVM IR metadata encodes the type relation as a graph of `DIType` nodes. All `DIType` nodes contain fields that can be filled with the type's bit-size, alignment, name and declaration coordinates. There are five subclasses of `DIType`: (1) `DIBasicType` contains an `encoding` field that has the same meaning as the DWARF `DW_AT_encoding` attribute. A `DIBasicType` node can thus be trivially translated into a `DW_TAG_base_type` DIE. (2) `DIDerivedType` nodes contain a `base_type` field which links to another `DIType` node and a `tag` field that specifies the tag of an equivalent DWARF DIE. Depending on the tag, there are two possible meanings a `DIDerivedType` can convey. First, it can describe a type derived from the `base_type`, such as a type alias, a reference or pointer to the base type, or a const-qualified version of it. The kind of derived type is provided via the tag. Second, `DIDerivedType` nodes are also used to describe members of composite types. These members can be data members, friend types, inherited types and static variables. The member kind can be obtained via the tag. (3) The composite types are described by `DICompositeType`, which provides an `elements` field that points to an array which includes references to its members described by `DIDerivedType` nodes and also nodes encoding declarations for member functions. (4) `DISubroutineType` nodes encode the type of a function and contain a `types` field that links to a type array. The array's first element is a reference to the return type of the respective function and the remaining elements are references to the types of the parameters. (5) The last subtype, `DIStringType` is used for Fortran programs only and therefore not within the scope of this thesis.

Although `DIType` is a subclass of `DIScope`, only nodes of `DICompositeType` act as scopes. Also, the scope fields of `DIBasicType` and `DISubroutineType` are always set to a null reference and `DIDerivedType` provides a scope only for type aliases.

The absence of a type is encoded as a null reference. Thus, LLVM IR producers describe the C pointer type `void*` as a `DIDerivedType` with `tag` set to `DW_TAG_pointer_type` and `base_type` set to `null`. Also, the return type of a function returning `void` should be provided as `null` in the type array of `DISubroutineType`. Contrary to the description of `DISubroutineType` in the LLVM Language Reference Manual [10], we found that there are also instances, in which Clang omits the return type `null` from type arrays for functions that do not have parameters and return `void`.

4.2.3 Parameters and Variables

If the intermediate representation for a function in LLVM provides debugging information, the function instance references a `DISubprogram` node, which is also an instance of a `DIScope` node. As mentioned in Section 3.2, `DISubprogram` nodes contain `file` and `line` fields. These specify the declaration coordinates. A column is not provided, though. Additionally, a `DISubprogram` node provides the name and linkage name of the function.

The type of the function can be obtained in two ways: First, each `DISubprogram` has a `type` field that links to a `DISubroutineType`. Second, for each parameter, a `DILocalVariable` node exists that provides the type reference via its `type` field. The `DILocalVariable` node additionally provides the name and declaration coordinates of the parameter. `DILocalVariable` nodes are also used to describe the attributes of local variables that are not parameters. The two cases can be distinguished by the `arg` field, which is zero for non-parameter variables and positive for parameters. In the latter case it describes the one-indexed position in the parameter list of the subprogram. The subprogram can be reached from the `DILocalVariable` by dereferencing `scope`. The reverse direction is not always possible: Although in the description of `DILocalVariable`, the LLVM Language Reference Manual [10] requests that all nodes describing parameters be included in the `retainedNodes` array field of their corresponding `DISubprogram`, our tests showed that consumers cannot rely on that behavior, as Clang does not add parameters to `retainedNodes` in unoptimized builds. Apart from the retained nodes, `DISubprogram` nodes do not feature references to their formal parameters.

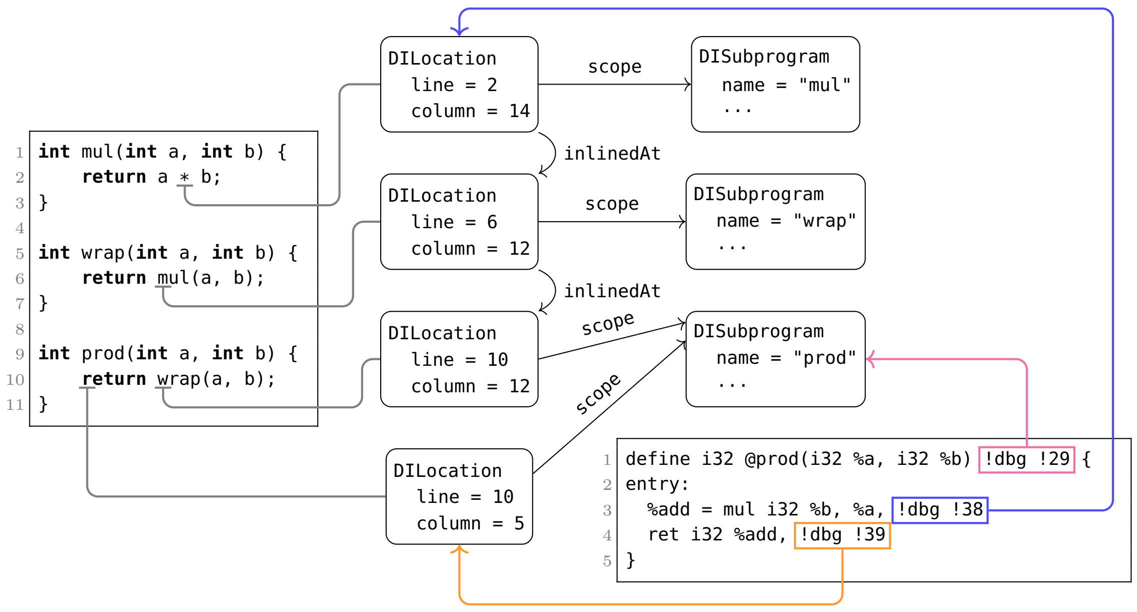
4.2.4 Inlining

Instructions that are result of inlined function calls feature a non-null `inlinedAt` field in their attached `DILocation`. `inlinedAt` references another `DILocation` that describes where the inlined call occures. This `DILocation` may in turn also feature a non-null `inlinedAt` field, if an inlined call occurres within an inlined function. Figure 4.3 shows such an example of nested inlining. The scope where the outermost inlined call occurred is reachable by repeatedly following `inlinedAt` until it is a null reference and then dereferencing `scope`.

For each inlined call, only one node exists that describes its location. Also, if two inlined calls occur at exactly the same source location—that means in the same file and at the same line and column—and are part of the same function or inlined function, LLVM does not combine their `DILocation` nodes despite them sharing the same content. Thus, inlined calls can be identified by their `DILocation` node.

4.3 Implementation in TPDE

In contrast to the line table, we perform the construction and serialization of the tree of DWARF DIEs in two separate steps. The following properties of the DWARF encoding



```
int mul(int a, int b) {
    return a * b;
}

int wrap(int a, int b) {
    return mul(a, b);
}

int prod(int a, int b) {
    return wrap(a, b);
}
```

```
define i32 @prod(i32 %a, i32 %b) !dbg !29 {
entry:
  %add = mul i32 %b, %a, !dbg !38
  ret i32 %add, !dbg !39
}
```

DILocation line = 2 column = 14
scope
DISubprogram name = "mul" ...
inlinedAt
DILocation line = 6 column = 12
scope
DISubprogram name = "wrap" ...
inlinedAt
DILocation line = 10 column = 12
scope
DISubprogram name = "prod" ...
scope
DILocation line = 10 column = 5

Figure 4.3: *Simplified LLVM IR featuring nested inlining produced by `clang -g -O2` for the source on the left. Metadata nodes are displayed as graph. For clarity, all nodes except `DILocation` and `DISubprogram` nodes are removed. The inlined call to function `wrap` in `prod` inlines a call to `mul`. The multiplication instruction therefore references a `DILocation` whose `scope` is the `DISubprogram` `mul` and the scope of the outermost inlined call is `prod`.*

hinder direct emission of the tree: (1) Adding a DIE into an already serialized tree would require moving all DIEs that succeed the inserted DIE in pre-order traversal. (2) The encoded length of a DIE is not fixed, because it may be altered when adding and removing attributes and, in the case of variable-length encodings, changing their values. Thus, also modifications of DIEs can require moving other DIEs.

Since these moves within memory can occur a significant runtime penalty for larger trees, we consider direct emission of DWARF DIEs only feasible for producers which do not have to modify any parts of the tree once generated. We do, however, note that direct emission could still be employed for a subset of the tree which the producer knows of that it does not change during further compilation. For TPDE, we do not use this optimization.

4.3.1 Tree Data Structure

During the compiler passes, we construct an *intermediate tree*, which is an own data structure that resembles the DWARF tree of DIEs. Then, after all functions have been compiled, the intermediate tree is serialized into a tree of DIEs.

The nodes of the intermediate tree are bump-allocated within a dynamic-sized byte-array. If needed for alignment, they are preceded by padding bytes. The size of each node depends on its *payload*. The one-byte *payload type* specifies the kind of the payload. It also implies the DWARF tag of the resulting DIE and specifies whether the DIE can have children. The reverse does not hold: Multiple DIEs with the same tag can result from nodes with different payload types. For example, DIEs with tag `DW_TAG_formal_parameter` are both used for the formal parameters of functions and of function *types*. In the former case they are children of `DW_TAG_subprogram` DIEs and in the latter case they are children of `DW_TAG_subroutine_type` DIEs and contain only a `DW_AT_type` attribute. To save memory in the latter case, we use a different payload that only contains a field for the type reference.

In contrast to encoded DIEs, the fields in the payloads have a fixed size. While this enables TPDE to change attributes of nodes after creation, it also imposes limits on the range of their values. The sizes of the fields are chosen to meet realistic use cases; for extreme values, our choices can be adjusted via template parameters.

In addition to their payloads and payload types, all nodes also feature two references to other nodes. The references are realized as indices into the byte-array which contains the intermediate tree. The first reference links a node to its directly succeeding sibling. If there is no such sibling, then the offset is zero. Although zero is also a valid offset to a node, there is no conflict, because TPDE ensures that the node at offset zero is always the root node, which cannot be a sibling of another node. The second reference links a node to its first child node if it has children. The first child node thus acts as the head of a linked list of all children, which is formed by the references to the sibling nodes.

While this data structure offers trivial pre-order traversal of the tree, which is needed for the serialization, it only allows insertion of new nodes in constant time at the beginning of a child list. We avoid this problem by performing insertions of child nodes in reverse order for all child nodes that have ordering requirements, such as formal parameters and data members of composite types. Once emplaced into the byte-array, nodes are not freed until the end of compilation. Also, nodes are never reparented: Once a node has been added to the child list of its parent node, it remains part of this list. The data structure does not provide any means to obtain a reference to the parent node from a child node.

4.3.2 Tree Construction

Before starting the compilation, TPDE emplaces the root node describing the compilation unit at offset zero in the byte-array. It uses information it obtains from a `DICompileUnit` node that is referenced by the LLVM IR `Module` object. There are following additional locations in LLVM IR, where TPDE finds references to debugging information metadata: (1) Instructions can contain references to a `DILocation` as described in Sections 3.2 and 4.2.4, (2) Functions can link to a `DISubprogram` and (3) there are *debug records*, which appear interleaved with instructions and contain references to `DILocalVariable` and `DILocation` nodes to provide information about the variable's values. All other debugging information metadata is indirectly accessed via these references.

During compilation, TPDE appends a corresponding node to the intermediate tree whenever it first encounters a `DINode`. Additionally, TPDE adds an entry into a hash map that links from the `DINode`'s address to the offset of the created node. There are multiple maps for different subtypes of `DINode`. The maps are used to prevent duplicate generation of nodes for the same entity and to obtain offsets of already generated nodes.

The latter use case is especially relevant for scopes: After emplacing a node generated from a `DINode` that features a `scope` field, the newly emplaced node must be added to the tree by putting it into its parent's child list. The parent is the node generated from the `DINode` linked to by `scope` or the root node if `scope` refers to global scope. It is not guaranteed that the parent node already exists. Thus, the pointer provided by `scope` is first used to look up the offset of a possibly existing node. If the loookup succeeds, the obtained offset is used. Otherwise, a function is called that translates the `DIScope` node to a corresponding node in the intermediate tree.

Offsets to other nodes also need to be known when constructing nodes that contain references to other nodes. These references are encoded in the same way as the references to siblings and the first child. The most common are type references, as found for example in derived types, variables, and, for the return type, in subprogram nodes.

When creating a node that references another node, the referenced node is in most cases looked up or generated before the referring node is emplaced. However, to break recursive call cycles resulting from cycles in the type relation, we added cases in which TPDE first emplaces the referring node with a placeholder reference. TPDE later replaces the placeholder with the correct reference after registering the referring node in the corresponding map.

Apart from resolving references, the translation of `DINode` s primarily consists of copying contents of relevant field to the nodes in the intermediate tree. This also includes the simultaneous construction of the `.debug_str` section: Strings are directly copied into that section and—similar to `DW_FORM_strp`—only the string's offset is saved in the intermediate node. We do not take any measures to prevent duplicate entries in `.debug_str`.

Nodes of type `DICompositeType` require additional processing as the nodes referenced in their `elements` field have to be translated, too, and must be appended as child nodes in reverse order. Another type of node that requires additional care is `DISubprogram`. Its formal paramters must be appended in reverse order. Thus, it is not sufficient to append these on first encounter. Also, the `DILocalVariable` nodes for formal parameters are not always referenced in the `retainedNodes` array, which furthermore does not guarantee any order. Since we cannot obtain references to the formal parameters in correct order, we instead rely on the information from the subroutine's `DISubroutineType`. Thus, TPDE creates a formal parameter node for each parameter type in the `types` array with placeholders for all fields except the type reference. Later, when TPDE encounters the `DILocalVariable` node for the parameter, it fills the missing information. Because TPDE always translates the `DISubprogram` node before it compiles the corresponding function, it never encounters a `DILocalVariable` for a parameter node before it creates the intermediate node.

However, we found examples, in which Clang generates nodes for formal parameters whose types are not included in the `types` array of the corresponding `DISubroutineType`. Because this affects only one specific compiler-generated function, we assume that this behavior is not intended. We reported this issue to the LLVM developers[3] and applied a temporary fix in our evaluation builds.

Because there are no nodes for inlined instances of `DILocalScope` and `DILocalVariable`, TPDE identifies these by a pair of addresses. The first address is that of the `DILocalScope` or `DILocalVariable` and the second is that of the `DILocation` corresponding to the inlined

[3]`https://github.com/llvm/llvm-project/issues/104765`, accessed 2024-08-20

function call. Additional maps exist that use these pairs as keys. Whenever TPDE encounters a `DILocation` with a non-null `inlinedAt` reference, it looks up or recursively creates nodes for both the inlined scope and the scope it is inlined at.

When TPDE compiles a function, it does not yet know whether it will later encounter an inlined instance of that function. Thus, we decided not to encode abstract instance trees in the intermediate tree. Instead, we store both the information for abstract DIEs and the out-of-line DIEs in regular subprogram nodes. These regular nodes are referenced by inlined instances of the function via `abstract_origin`. Whenever TPDE encounters an inlined instance of a function, it flags the node of the original function as containing both the abstract and the out-of-line instance.

4.3.3 Ranges

The local scope nodes for inlined and non-inlined lexical blocks and for inlined subroutine calls features an offset into an array containing multiple range lists. These are organized as circular singly-linked lists of entries each containing a start and an end address. The lists are circular so that one reference in the local scope node provides fast access to both the beginning and the end of the list.

During compilation, TPDE remembers the values of `scope` and `inlinedAt` of the `DILocation` linked by the last LLVM instruction. Whenever these values are different for the next instruction, TPDE *finishes* the current range and *starts* a new one. Starting a range only means saving the address of the next machine code instruction. When TPDE finishes a range, it looks up the node for the scope described by the saved `scope` and `inlinedAt` fields. Then it traverses the node and its parent nodes until it reaches a non-local scope node. For this purpose, the payloads of local scope nodes feature a reference to their parent nodes. For each of the visited nodes, TPDE appends a range entry to the nodes range list with the saved start address and with the current address of the next machine instruction as end address. TPDE coalesces the new range entry with the previous last range entry, if there is no gap in between them.

After compiling a function, TPDE traverses the subtree rooted at the function's subprogram node. For each local scope node, it serializes its range list into the `.debug_rnglists` section. It then replaces the offset into the range list array used during construction with the offset of the serialized range list in the `.debug_rnglists` section. This offset constitutes the value for the `DW_AT_ranges` attribute. Afterwards, the contents in the range list array are not needed anymore and the array can thus be reused when compiling the next function.

The program counter range that belongs to a `DISubprogram` is also tracked, but in a more simple manner: Because it is guaranteed to be contiguous, it is sufficient to record the write pointer offset in the code buffer before and after compiling the function. The resulting values are used for the `DW_AT_low_pc` and `DW_AT_high_pc` attributes, respectively.

4.3.4 Tree Serialization

After the last compiler pass has run for the last function and its range list is serialized, TPDE serializes the intermediate tree into the `.debug_info` section. For that, it uses the references to the siblings and to the first children to traverse the tree in pre-order. When serializing a node, TPDE saves the offset of the resulting DIE in the `.debug_info` section in the header of the node. This saved offset is used as the value for references of other DIEs to this DIEs. When the serialized offset of a referred to DIE is already known during serialization of a node, this offset is directly written into the buffer for the `.debug_info` section. All occurences of forward references to not-yet serialized DIEs are saved and the references are filled with a placeholder value. Because reference offsets have a fixed size, patching them after tree serialization is possible. The references to siblings and the first children are not serialized as their informational content is conveyed via the layout of the DIEs in the `.debug_info` section.

When TPDE encounters a subprogram node that is flagged as representing both an abstract and out-of-line instance, the subtree starting at that node is serialized twice. In the first traversal of the subtree, only the attributes relevant for the abstract instance tree are included. In the second traversal, the set of attributes is inverted and an additional `DW_AT_abstract_origin` attribute is emitted. The value of this attribue is DIE's serialized offset as saved in the previous traversal. The second traversal does not overwrite the saved serialized offsets, so that all nodes in inlined instances link to the abstract DIEs as opposed to out-of-line DIEs.

TPDE generates abbreviations when these are needed to serialize a specific node. Using template programming, we created one function for each payload type that serializes suitable abbreviations. The boolean conditions in nodes that influence the generation of abbreviations are extracted via dedicated functions and together with the payload type they form a `payload key`. This payload key uniquely identifies an abbreviation. It is used as an index into an array that contains the abbreviation code of each abbreviation that is already serialized. Thus, for every node that is serialized, the abbreviation key is generated and used to look up the abbreviation code. If the lookup returns zero, then TPDE generates the corresponding abbreviation and saves the resulting code in the array.

In the course of our work, it showed that cases in which there are multiple possible abbreviations for one payload type can almost entirely be avoided, except for following nodes: First, nodes referring to types which can be `void` need one abbreviation with `DW_AT_type` and one without the attribute. Second, base types nodes contain a size description which can either be in bytes or in bits. Because debuggers refused to treat bit-sizes that were multiples of eight in the same manner as corresponding byte sizes, we emit `DW_AT_byte_size` when the bit-size is a multiple of eight and otherwise `DW_AT_bit_size`. With reasonable effort, we could use different payload types for void and non-void variants of nodes as well as bit-sized and byte-sized variants. Then we could use the payload type directly instead of the abbreviation key for the lookup.

Additionally, we could hardcode the complete `.debug_abbrev` section, and use abbreviation keys or payload types directly as abbreviation codes. However, because TPDE can currently only generate 60 different abbreviations, we do not expect that these optimizations would significantly improve performance.

4.4 Limitations

Due to the limited scope of this thesis, we do not handle all information available to TPDE that can be encoded in the `.debug_info` section. In particular, we do not read the field `templateParams` of `DISubprogram` nodes and thus do not emit template information. Also, our current version of TPDE does not create DIEs describing the positions of labels, which it could obtain from LLVM IR through `#dbg_label` records.

In addition, both DWARF and LLVM feature mechanisms to link from one compilation unit to contents of another. However, TPDE only supports references within compilation units and does not read the `imports` field of `DICompileUnit` nodes. Furthermore, we do not emit *call site information* except for inlined calls. Call site information can be used by debuggers to locate function calls and obtain their attributes if the debugger cannot derive these from the machine instruction sequence.

5 Variable Locations

Debuggers not only provide the possibility to track control flow by showing the program counter's position in terms of source code. They also allow users to inspect the state of the program when it is paused. This state primarily consists of the contents of registers and memory. Debugging information formats can provide descriptions that map the state of registers and memory to values of source-level variables.

5.1 Representation in DWARF

DWARF provides variable location information through *location descriptions*, which are implemented as *expressions* and *location lists*.

5.1.1 Expressions

Each expression provides the value of a variable or its location in memory and registers. An expression consists of a sequence of byte-code operations preceded by an LEB128-encoded number specifying the length of the sequence in bytes. A DWARF consumer evaluates it by executing it on a stack machine.

Each operation consists of a one-byte opcode followed by operands. In the following, we provide an overview of operations that are defined by DWARF. Our list only includes operations relevant to this thesis. For an exhaustive enumeration, we refer the reader to the fifth version of the standard [2].

Constants Various operations push constants onto the stack. The operations differ in the encodings of their operands. There are dedicated operations that push small integer values and only occupy one byte.

Stack Operations DWARF defines operations that add, remove or reorder elements on the stack. Examples include `DW_OP_dup`, which duplicates the top of the stack and `DW_OP_swap`, which swaps the topmost two elements of the stack.

Memory and Register Access The operations `DW_OP_breg0` to `DW_OP_breg31` load the contents from the register with the respective number, add an offset and push the result onto the stack. The numbering of the registers is defined externally by the Application Binary Interface (ABI) of the architecture. For x86-64 platforms that adhere to the System V ABI, the register number mapping can be found in [6]. In addition, DWARF defines `DW_OP_fbreg`, which loads the base address of the stack frame instead of a register value. The value or location of the stack frame base address is encoded in the `DW_AT_frame_base` attribute of the subroutine DIE which describes the function the stack frame belongs to. Memory access works through `DW_OP_deref`, which loads an address-sized value from the memory location specified by the top of the stack. It then replaces the top of the stack with the loaded value.

Memory and Register Locations By default, expressions provide memory locations. Thus, a DWARF consumer interpretes the top of the stack after expression evaluation as the address of the variable's value in memory. A typical example for memory locations are that on the program's stack. They can be specified using only one `DW_OP_fbreg` operation, which pushes a variable's absolute address onto the DWARF stack when the frame offset of the variable is given as operand. An expression terminated by `DW_OP_stack_value` describes the variable's *value* instead of its address.

Register locations are encoded using the operations `DW_OP_reg0` to `DW_OP_reg31`. There is a difference between specifying a location and providing an expression that evaluates to the contents of that location: In both cases, the debugger can access the value of the variable by fetching the respective register and memory contents. However, expressions that specify locations additionally enable the debugger to modify the values and, in the case of memory locations, provide the address of a variable to the user. For example, on x86-64, a producer can describe a value located in register `RSI`, which has number 4, either with `DW_OP_reg4` or `DW_OP_breg4 0, DW_OP_stack_value`. The former additionally describes a location and should therefore be preferred.

Arithmetic and Logical Operations DWARF defines unary and binary operations that operate on the topmost stack entries. Two of the most common are `DW_OP_plus` and `DW_OP_and`. The former is particularly useful to calculate a pointer as a sum of a base address and a relative offset, such as the offset of a field within its structure type. `DW_OP_and` is frequently used to limit the size a value obtained by `DW_OP_deref` by applying a bit mask.

Control Flow Expressions can also contain jumps and branches to both previous and subsequent operations. Thus, conditional execution and loops are possible, which allows for arbitrary computation. Oakley [7] demonstrated this capability for DWARF expressions employed in stack unwinding information that certain implementations of exception handling depend on.

Types Starting with DWARF version 5, stack entries can also contain values whose types are different from the type of memory addresses. There are variations of other operations which additionally require an operand that is a reference to a base type DIE specifying the result type. Furthermore, stack entries can be converted between different types using dedicated operations. The types not only influence the size of values loaded from memory via the `encoding` attribute of the base type DIE, they also affect the semantics of arithmetic operations. Thus, `DW_OP_plus` performs an integer addition if executed on two integers and a floating point addition if executed on floating point values.

Composite Locations When a variable's value is split into mutliple pieces, the location of each piece can be specified independently. A producer describes such composite locations by appending `DW_OP_piece` or `DW_OP_bit_piece` operations to the ends of the expressions that describe the values or locations of the pieces. The first operand of `DW_OP_piece` and `DW_OP_bit_piece` provides the size of the piece in bytes or bits, respectively. `DW_OP_bit_piece` additionally takes a second operand, which specifies the position of the piece inside the value described by the expression. The resuling operation sequences of all pieces are then concatenated in the order of the pieces. Thus, on little-endian architecures, the piece containing the least significant byte appears first in a composite location description. Following expression illustrates the semantics of `DW_OP_piece`:

```
DW_OP_reg0, DW_OP_piece 8, DW_OP_constu 0xdeadbeef, DW_OP_piece 4
```

On x86-64, this expression encodes that the lower 8 bytes of a 12-byte variable are located in register 0, which is `RAX`, and the remaining bytes have the constant value `0xdeadbeef`.

Variables are linked to their location descriptions via the attribute `DW_AT_location`. In combination with the form `DW_FORM_exprloc`, expressions can be used directly as values of `DW_AT_location`. Empty expressions encode the absence of location and value information.

5.1.2 Location Lists

When variables are held in registers, the register allocations are typically subject to change since the amount of available physical registers is limited and the set of variables used for computation changes during program execution. Thus, the variable locations can depend on the program counter. However, DWARF expressions provide no means to read the program counter. Instead, DWARF offers *location lists*. These lists are similar to range lists, but additionally provide an expression per range entry. Using location lists, a DWARF producer can specify multiple locations or values per variable. Each expression is only valid within its associated program counter range.

The location lists are stored in the `.debug_loclists` section. Each contribution to `.debug_loclists` starts with a header that indicates the length of the contribution and provides additional information needed for decoding. The header is followed by the encodings for the location lists, which are sequences of location list entries. Each location list entry either provides a program counter range with an expression, a base address used for relative encodings of subsequent ranges or a default expression without an associated range. A default expression describes the location of a variable for positions in the program that are not covered by any of the other entries. Each list is terminated by an end-of-list entry. Table 5.1 shows that the entries in location lists do not have to be sorted by their range. As opposed to the ranges of range list entries, those of location list entries may intersect, in which case the entries supply the debugger with multiple simulataneous locations.

The encoding of a list entry starts with a one-byte value indicating the entry's kind and—if applicable—the encoding of the range. The options for the encoding are similar

Table 5.1: *Example of a location list. The variable is stored in the registers with numbers* 4 *and* 0*, respectively, when the program counter is within ranges* $[0, 10)$ *and* $[20, 25)$*. Otherwise, the variable is located on the stack.*

Entry Kind	Range	Expression
Range with expression	[20,25)	`DW_OP_reg0`
Default location	-	`DW_OP_fbreg 8`
Range with expression	[0,10)	`DW_OP_reg4`
End of list	-	-

to those offered for range lists. If the entry contains a range, the encoding of the range follows the start byte, and if it features an expression, the expression encoding comes last.

If a location list is used for a variable, then the value of its `DW_AT_location` attribute is the offset of the location list within the `.debug_loclists` section. Alternatively, DWARF provides the option to store the offsets of the location lists in an array after the `.debug_loclists` section header. This enables producers to provide the location list reference as an index into that array, which the producer indicates by using a dedicated form.

5.2 Encoding in LLVM IR

The final locations of local variables are an artifact of code generation. Thus, they are not part of LLVM IR, which is used to describe functions before instruction selection and thus also prior to register allocation. LLVM IR therefore describes variable locations in terms of SSA values. [11]

Within each basic block, SSA values do not change their contents. A static mapping from SSA values to variable values would transfer this property to the latter. However, variables in imperative languages such as C and C++ are a stateful concept. Even within a part of source code that results in one single basic block, there can be multiple assignments to one variable.

Classically, producers of SSA-based intermediate representation thus employ *variable renaming*: Each assignment to a variable in the source code results in an assignment to a new SSA value in the intermediate representation, so that each value is only assigned to once. [14]

Because of the memory access instructions, frontends generating LLVM IR do not have to implement variable renaming themselves. Instead, they can allocate stack space for local variables in the entry block of a function and describe assignments to variables as `store` instructions. However, `store` instructions complicate optimizations because of their side effect in memory. Thus, there is the LLVM pass `mem2reg` that performs a task similar to variable renaming: Instead of assignments to source-level variables, it tracks stores to locations on the stack and creates corresponding SSA values[1].

[1]Source: `lib/Transforms/Utils/PromoteMemoryToRegister.cpp`

5.2.1 Debug Records

LLVM IR provides the relationship between variables and SSA values through *debug records*. These appear interleaved with instructions in basic blocks. Each debug record used to describe variable locations features a reference to the corresponding `DILocalVariable` metadata node of the variable. Furthermore, debug records can also contain a reference to a `DILocation` node. This reference is particularly relevant in the context of inlined functions: When the `DILocation`'s `inlinedAt` field is non-null, the debug record describes the location of the inlined instance of the variable described by `DILocalVariable`. There are no separate `DILocalVariable` nodes for inlined variable instances.

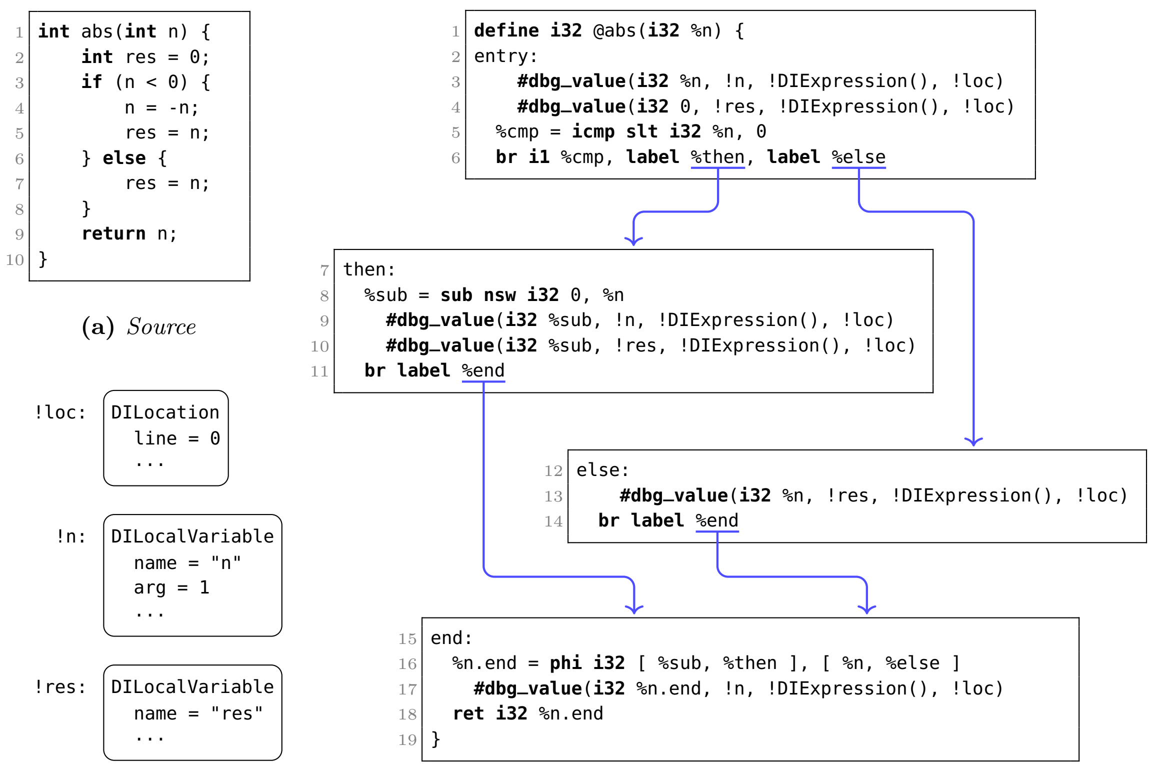
To describe the relationship between a variable and its location in memory, LLVM features the `#dbg_declare` record. It references an `llvm::Value`, which provides the address of the variable. The `llvm::Value` is either a constant, an SSA value or a function argument. In `#dbg_declare` records, it is typically produced by an `alloca` instruction in the entry block which allocates the space for the variable on the stack. In these cases, the `#dbg_declare` record often appears in the entry block. For each concrete instance of a variable, all `#dbg_declare` records must describe the same location [11]. Thus, `#dbg_declare` records describe a static relationship between variables and SSA values. This is sufficient for local variables statically allocated on the stack: Since assignments are performed through `store`, the value seen by the debugger is automatically updated when the value changes in the program, even though the location remains the same.

However, there is not always a one-to-one relationship between a variable and a single storage location in memory. For example after `mem2reg`, different SSA values represent the value of a variable at different points in the program. Such cases are expressed with `#dbg_value` records, which associate an `llvm::Value` with a variable at one point in the program. There can be multiple `#dbg_value` records that link the same concrete variable instance to different values. Although `#dbg_value` records describe the variable's values, their usage does not imply the emission of DWARF expressions which do not provide variable locations. The location information is implicitly provided in `#dbg_value`: The location of the variable is the same as the location of the `llvm::Value` as determined by register allocation.

Conceptually, a `#dbg_value` record can be seen as an assignment to an imaginary register that tracks the `llvm::Value` of the referenced variable. This assignment is always performed before the instruction below the record is executed. Because any instruction in a basic block is only executed after all preceding instructions have been executed, one can infer the variable's current `llvm::Value` for each instruction by finding the closest preceding `#dbg_value` record that matches the variable. If there is, however, no such `#dbg_value` record, the value depends on the control flow of the program, which is not fully known during compile time. An example for this situation is depicted in Figure 5.2.

In addition to `#dbg_value` and `#dbg_declare`, recent releases of LLVM starting with version 17.0.1 also feature an experimental `#dbg_assign` record[2]. Where possible, this record describes both the address and the value of a variable. Hence, a `#dbg_assign` record features the fields of both other mentioned record types. Furthermore, it links to a

[2]First proposed in `https://discourse.llvm.org/t/rfc-assignment-tracking-a-better-way-of-specifying-variable-locations-in-ir/62367`, accessed 2024-08-28



```
int abs(int n) {
    int res = 0;
    if (n < 0) {
        n = -n;
        res = n;
    } else {
        res = n;
    }
    return n;
}
```

(a) *Source*

```
!loc:   DILocation
          line = 0
          ...

!n:     DILocalVariable
          name = "n"
          arg = 1
          ...

!res:   DILocalVariable
          name = "res"
          ...
```

(b) *Metadata nodes*

```
define i32 @abs(i32 %n) {
entry:
    #dbg_value(i32 %n, !n, !DIExpression(), !loc)
    #dbg_value(i32 0, !res, !DIExpression(), !loc)
  %cmp = icmp slt i32 %n, 0
  br i1 %cmp, label %then, label %else

then:
  %sub = sub nsw i32 0, %n
    #dbg_value(i32 %sub, !n, !DIExpression(), !loc)
    #dbg_value(i32 %sub, !res, !DIExpression(), !loc)
  br label %end

else:
    #dbg_value(i32 %n, !res, !DIExpression(), !loc)
  br label %end

end:
  %n.end = phi i32 [ %sub, %then ], [ %n, %else ]
    #dbg_value(i32 %n.end, !n, !DIExpression(), !loc)
  ret i32 %n.end
}
```

(c) *Basic blocks*

Figure 5.2: *Simplified LLVM IR showing* `#dbg_value` *semantics. In block* `else`*, variable* `!n` *is known to have* `llvm::Value %n` *because the only predecessor,* `entry`*, provides it via* `#dbg_value`*. In contrast, the* `llvm::Value` *of* `!res` *in* `end` *is unknown at compile time since* `then` *and* `else` *assign different values to* `!res`*. We note that in this particular example, a possible optimization is to make* `!res` *an alias of* `!n`*. Then, a* `#dbg_value` *assigning* `%n.end` *to* `!res` *can actually provide a value for* `!res`*. The LLVM IR was obtained by compiling the source with* `clang -g -O0` *and applying the* `mem2reg` *pass.*

metadata node of type `DIAssignID`, which does not contain any content but has a distinct pointer and can thus be used to identify the source-level assignment which lead to the change of the variable's value. Instructions like `store` also link to the same `DIAssignID` if they are result of the same source-level assignment. A `#dbg_assign` record is only valid as long as the assignment it describes is still present in the intermediate representation. [9]

5.2.2 Expressions

To handle cases in which the value of a variable cannot *directly* be described by an `llvm::Value`, `#dbg_value` also references an expression that transforms the value. Analogously, also `#dbg_declare` records feature expressions, which transform the addresses.

The expressions are implemented as `DIExpression` metadata nodes, each of which comprises an operation sequence whose instruction set is derived from that of DWARF expressions [10]. Not all of the DWARF operations can occur in these nodes: While they support memory access instructions, register access is not possible. Furthermore, `DIExpression` nodes do not support all of the arithmetic and stack instructions and connot contain control flow operations. While `DIExpression` nodes do not support the DWARF type conversion instructions, LLVM provides a custom `DW_OP_LLVM_convert` operation, in which the reference to the base type DIE is replaced by operands that contain the values for `size` and `encoding`.

Also, the operations `DW_OP_piece` and `DW_OP_bit_piece` are not part of the `DIExpression` vocabulary. LLVM IR instead describes variable values composed of multiple pieces by multiple debug records. The expression in each such record ends with a `DW_OP_LLVM_fragment` operation that specifies the offset and size of the piece within the variable's value.

Expressions also allow location descriptions derived from multiple `llvm::Value`s. If a debug record only references one value, it is implicitely pushed onto the stack before the—possibly empty—expression is executed. If it contains references to multiple values, then these are instead explicitely retrieved via `DW_OP_LLVM_arg`.

5.3 Implementation in TPDE

TPDE performs register allocation in conjunction with instruction selection and code emission. It keeps all necessary state in data structures that are cleared after each function. Additionally, the register assignment is stored in arrays that only reflect the current state. Thus, there are no means to reconstruct the location of an SSA value if it is not traced during the last compiler pass. Therefore, we also integrated the generation of variable location information into this compiler pass.

5.3.1 Location List Data Structure

Due to the limited scope of this thesis, we always provide variable locations through DWARF location lists. Our location lists do not contain entries that specify a default location or a base address for relative range encodings. Thus, all entries except of the end-of-list entries feature a range and an expression.

The location lists are constructed in a dedicated dynamic-sized byte array. For each location list entry, a header is bump-allocated in the byte array, preceded by padding if necessary for alignment. The header contains the start address of the range and the length of the expression's encoded instruction sequence, which follows the header in the byte array. Additionally, it contains an offset that points to the header of the next entry. Via this offset, the headers form a circular linked list for each location list similar to the data structure which encodes range lists. Each local variable or parameter node in the intermediate tree stores a reference to one such circular linked list.

The headers do not provide the end addresses of their respective ranges. However, we enforce that entries are ordered by their start address and that there are no gaps between the ranges. Thus, the end address equals the start address of the next entry's range. If there is no next entry, the end address is that of the containing function's program counter range. To prevent gaps, we handle ranges without location information by storing a header with an empty operation sequence.

After compilation of a function is finished, TPDE serializes the location lists for each variable or parameter into the `.debug_loclists` section. For that it serializes the range and expression length for each header and copies the encoded operation sequence. It skips all headers that specify an empty range or expression. After serialization, the location list reference in the intermediate node is modified to provide the offset of the serialized location list within the `.debug_loclists` section.

5.3.2 Location List Construction

To infer the final locations of variables, the information provided in debug records has to be combined with the locations of SSA values, which are found in the register assignment array and in further data structures TPDE uses. This task primarily consists of two subtasks: (1) We have to identify when our sources of information change. This means that we have to find instants in which the location of an SSA value changes or a debug record supersedes a previous record. (2) After each detected change, we must update the affected location lists accordingly.

Identification of Changes For each variable instance, TPDE keeps a reference to the last matching debug record. This reference is updated whenever TPDE encounters a debug record referencing the variable. TPDE appends a new entry to the location list when that reference changes. Additionally, TPDE tracks the set of SSA values that the debug record depends on so that it can also append a location list entry when any location of such an SSA value changes.

For this purpose, TPDE maintains a mapping from SSA values to the dependent variables. We exploit that existing code in TPDE assigns indices to SSA values, which eliminates the need for hashing. Our mapping is an array that contains a list head for each SSA value at its respective index. The remaining list nodes follow after the sequence of list heads. When an entry of a list is removed, its corresponding list node is marked as invalid, but remains part of the list. Our design is based on the observation that the number of debug records referencing one single SSA value is typically low and most of the time zero or one. Thus, most lists only comprise their head node.

Within the functions of TPDE that compile single LLVM instructions, we only added few calls to functions related to debugging information. One such call occurs whenever an SSA value's location changes: Our function then looks up all affected variables in the aforementioned mapping and generates new location list entries. Because this call occurs for all SSA values regardless of whether variable locations depend on them, it is important that our mapping offers fast lookup.

Generation of Location List Entries Whenever TPDE has identified the necessity to append an entry to the location list, it first emplaces a header in the byte-array with the start address set to the address of the next machine code instruction. It then successively translates the debug record into the DWARF operation sequence following the header, resizing the byte-array as needed. Currently, TPDE does not support `DW_OP_LLVM_arg`, hence the translation process always starts with the generation of a description for the referenced `llvm::Value`.

Depending on the kind of debug record, TPDE either produces a description of the value or the location of the `llvm::Value`. For `#dbg_declare` it emits a description of the *value*, because the value of the variable's address is the location of the variable. For a `#dbg_value`, TPDE appends a description of the `llvm::Value`'s *location*. If, however, the `DIExpression` of the debug record is non-empty, TPDE instead emits a description for the *value*, because DWARF operations cannot operate on location descriptions. In this case, TPDE has to append `DW_OP_stack_value` before finishing the translation of the debug record to prevent the DWARF consumer from interpreting the result as the variable's address.

Currently, TPDE does not include functionality to track whether `DIAssignID`s are referenced from instructions. Thus, TPDE does not encode locations of variables described by `#dbg_assign` records as it cannot infer the validity of the latter. If we knew, however, which `#dbg_assign` records are valid, the way we handle them would depend on the information they provide. If they provided an address, we would process them similarly to `#dbg_declare` and otherwise like `#dbg_value`.

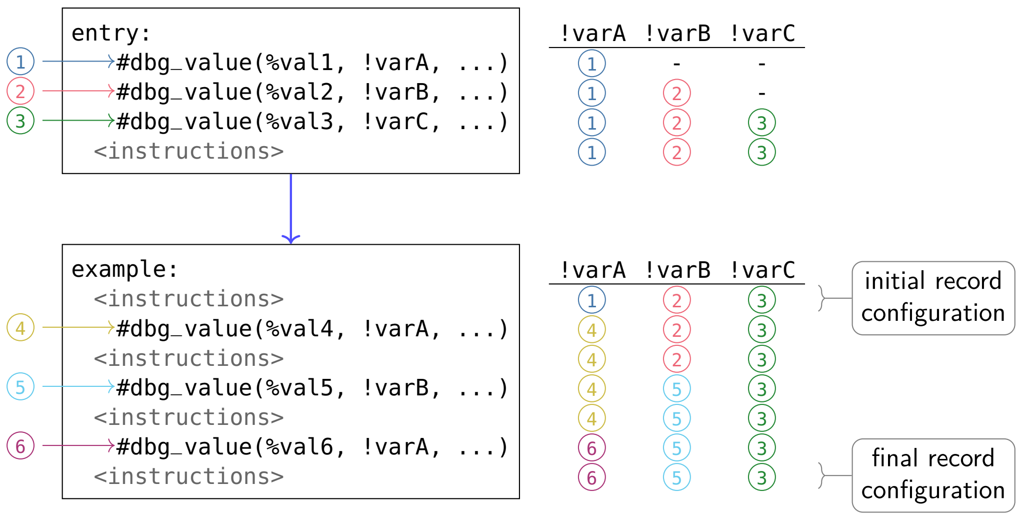
Following the operations generated for the `llvm::Value`, the encodings of the operations in the `DIExpression` are appended. Because of the limited scope of this thesis, we only implemented support for those operations that are defined by DWARF. Whenever TPDE encounters an instruction defined by LLVM, such as `DW_OP_LLVM_convert`, `DW_OP_LLVM_fragment` or `DW_OP_LLVM_arg`, TPDE *aborts* an expression. That means that it stops the translation of the debug record and sets the expression length in the location list entry's header to zero. This discards all operations already translated and thus ensures that no partial—and therefore wrong—expression is emitted.

In addition to unsupported operations in `DIExpression`, there are also unsupported kinds of `llvm::Value`s. This particularly includes constants which require relocation, such as pointers to global variables. Expressions are also aborted when TPDE cannot provide locations for `llvm::Value`s. This typically occurs when the lifetime of an `llvm::Value` has ended, which means that it is no longer used in the program.

5.3.3 Propagation of Debug Records

The approach we described in Section 5.3.2 is still incomplete and results in incorrect variable locations as it does not consider the propagation of `#dbg_value` records between basic blocks. In the following, we describe the rationale and the algorithm of an additional pass we added to TPDE to simulate the propagation.

In Section 5.2.1, we explained that `#dbg_value` records provide a dynamic mapping between variables and `llvm::Value`s that conceptually depends on the control flow of the executed program. In the remainder of this section, we call a debug record which describes a variable at one specific point in program execution the *active* record for the variable. In addition, we call the set of active records of all variables at the beginning and at the end of a basic block the *initial* and *final record configurations*, respectively. Our definitions are illustrated in Figure 5.3.



```
entry:
  #dbg_value(%val1, !varA, ...)
  #dbg_value(%val2, !varB, ...)
  #dbg_value(%val3, !varC, ...)
  <instructions>
```

!varA	!varB	!varC
1	-	-
1	2	-
1	2	3
1	2	3

```
example:
  <instructions>
  #dbg_value(%val4, !varA, ...)
  <instructions>
  #dbg_value(%val5, !varB, ...)
  <instructions>
  #dbg_value(%val6, !varA, ...)
  <instructions>
```

!varA	!varB	!varC
1	2	3
4	2	3
4	2	3
4	5	3
4	5	3
6	5	3
6	5	3

Figure 5.3: *A made-up example of two blocks with known control flow. Instructions are omitted. The corresponding active records for each variable are shown on the right.*

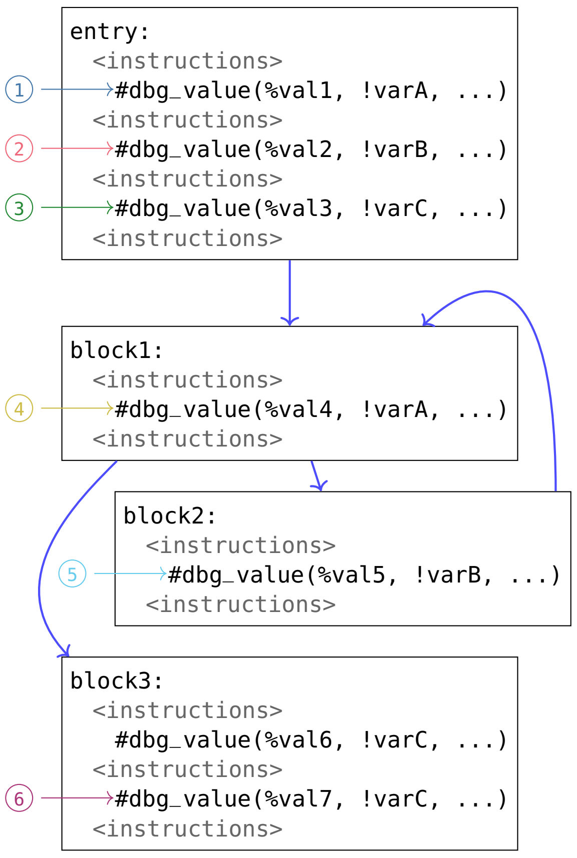
From any initial record configuration of a basic block, the compiler can derive the final record configuration by successively updating the configuration with the records found in the block. Thus, each basic block forms a *transfer function* for record configurations. We note that we do not distinguish between the different debug record kinds at this point, because all kinds have in common that they are active after their occurrence.

The initial record configuration of the entry block is trivially empty since control flow cannot yet have reached any debug record. For any basic block which has only one direct predecessor—such as `example` in Figure 5.3—the initial record configuration is the final record configuration of the predecessor. If a block has multiple predecessors—such as `end` in Figure 5.2, the initial record configuration depends on which predecessor transferred control flow. However, that is not known at compile time and furthermore not fixed during execution. Hence, the active record of a variable it only known, if the record configurations of all predecessors agree on the same active record.

We define a *combination function* which calculates the known subset of the initial record configuration of a basic block. This subset is the intersection of the known subsets of the final record configurations of all direct predecessors.

The transfer functions of the basic blocks and the combination function form an instance of a flow analysis problem. Our pass solves it using fixed-point iteration as outlined in [5]: Starting at the entry block, the pass computes the final record configuration for each block, visiting a block whenever its initial record configuration was updated. The combination function is used to update the inital record configurations of successors after evaluation of a block's transfer function. The pass terminates as soon as no record configurations change anymore when recomputed.

We split the implementation of the pass into three subpasses, which are executed prior to the former first compiler pass. In the first subpass, TPDE iterates over all debug records that belong to the current function to count the number of local variable instances and parameters. This is needed as we use data structures of fixed size in the



```
entry:
  <instructions>
    #dbg_value(%val1, !varA, ...)
  <instructions>
    #dbg_value(%val2, !varB, ...)
  <instructions>
    #dbg_value(%val3, !varC, ...)
  <instructions>

block1:
  <instructions>
    #dbg_value(%val4, !varA, ...)
  <instructions>

block2:
  <instructions>
    #dbg_value(%val5, !varB, ...)
  <instructions>

block3:
  <instructions>
    #dbg_value(%val6, !varC, ...)
  <instructions>
    #dbg_value(%val7, !varC, ...)
  <instructions>
```

(a) *LLVM IR*

Block	!varA	!varB	!varC
entry	①	②	③
block1	④	null	null
block2	null	⑤	null
block3	null	null	⑥

(b) *Transfer functions matrix*

Block	!varA	!varB	!varC
entry	①	②	③
block1	④	②	③
block3	④	②	⑥
block2	④	⑤	③
block1	④	null	③
block3	④	null	⑥
block2	④	⑤	③

(c) *Possible iteration order showing final record configurations calculated in each step*

Block	!varA	!varB	!varC
entry	null	null	null
block1	①	null	③
block2	④	null	③
block3	④	null	③

(d) *Result: Initial record configurations*

Figure 5.4: *LLVM IR example that includes a loop. If in the third subpass,* `block2` *is always pushed before* `block3`*, then* `block3` *is always handled directly after* `block1` *resulting in the iteration order shown in (c).*

next two passes. As a side effect, the first subpass also generates intermediate nodes for `DILocalVariable` nodes and their scopes. That is, because in this subpass, TPDE encounters `DILocalVariable` nodes for the first time if they were not referenced before by inlined instances of the function.

In the second subpass, TPDE extracts the transfer functions from the basic blocks. It encodes all transfer functions in a matrix which has a row for each block and a column for each variable. Each cell contains a pointer to the last debug record appearing in the corresponding block that references the respective variable. If the block does not contain a matching debug record, the cell stores a null pointer instead. Figure 5.4b shows the transfer matrix computed in the second pass for the blocks in Figure 5.4a.

In the last subpass, TPDE performs the propagation simulation. It maintains a stack to track the indices of the blocks whose final record configurations have to be recomputed.

Also, it utilizes an additional matrix which stores the initial record configurations. It has the same size and layout as the matrix storing the transfer functions. Variables that have no known active record at the beginning of a block are encoded as cells containing null pointers. Whenever TPDE has computed a final record configuration, it updates the initial record configurations of all its successors and, if it detects changes, pushes the successor's indices onto the stack if they not yet contained.

Updates of the initial record configuration of a successor are performed as follows: If the initial record configuration of the successor has not been computed before, the final record configuration is copied. Otherwise, the existing initial record configuration of the successor and the newly computed final record configuration are compared elementwise: In each column in which there are different pointers to debug records, the updated configuration receives a null pointer. Thus, after the first computation of an initial record configuration, the number of contained debug records can only decrease in subsequent computations.

One such situation in which an element of an initial record configuration is replaced with a null pointer is depicted in Figure 5.4c: After calculating the final record configuration of `block2`, the pass has to revisit `block1`, because the final record configurations of `block2` and `entry` differ in `!varB`. Thus, the second time `block2` is visited, its initial record configuration contains a null pointer for `!varB`, which in turn leads to revisiting of other blocks.

The matrix containing the initial record configurations—for our example it is displayed in Figure 5.4d—is the result of the propagation simulation pass. It is used during the last compiler pass: When TPDE begins compiling a block, it updates the saved references to the last debug records matching variables with the corresponding pointers found in the matrix.

In fact, we only perform the propagation simulation to detect where we have to drop variable location information. If LLVM IR featured information that specifies for each block, which variable locations are ambiguous due to different locations in the predecessors, we could avoid the propagation simulation pass. Instead, we would save the final record configuration after each compilation of a block. When starting the compilation of any block, we would reuse the saved configuration of an already compiled predecessor and drop all ambiguous locations. The compilation order of blocks guarantees that for each block except the entry block, at least one predecessor is compiled earlier.

6 Evaluation

In the previous chapters we described the different parts we added to the TPDE compiler back-end to enable generation of DWARF debugging information from LLVM IR. We will evaluate the quality of our implementation by comparing its output and runtime to that of LLVM's native back-end.

The choice of LLVM's back-end as a baseline is motivated by two factors: First, LLVM's back-end also consumes LLVM IR. Thus, we can ensure that deviations in our results are not caused by differences in the intermediate representation. Second, the LLVM back-end is natively supported by Clang, which allows us to use C and C++ programs as samples.

We will use a modified version of Clang that features TPDE as an alternative back-end and compile the C and C++ programs contained in the *SPECspeed 2017 Integer* [16] benchmarking suite. The measurement of variable location coverage will be performed on the *git* codebase.

Although compiling optimized LLVM IR is not an application the code generation of TPDE is designed for, we will perform our evaluation both for optimization levels `-O0` and `-O2`. This will enable us to also measure the impact of LLVM IR features that are primarily found in optimized code. We note that at `-O2`, Clang not only optimizes the intermediate representation, but also instructs the LLVM back-end to perform optimized code generation. We will not compile *x264* and *leela* at `-O2`, because the resulting LLVM IR utilizes features, such as floating point intrinsics that TPDE cannot generate code for.

We will start with a measurement of the variable location coverage and discuss the results. Next, a comparison of the impact of producing debugging information on compilation times will follow. We will finish with an examination of the DWARF section sizes in the produced binaries.

6.1 Local Variable Coverage

Local variable location coverage is an indicator of the quality of produced debugging information. Generally, it describes the fraction of the program for which information about variable locations is available. There are various metrics which define the *fraction of the program* differently.

LLVM includes the tool `llvm-dwarfdump`, which not only provides means to inspect DWARF debugging information contained in binaries, but also prints statistics including variable coverage when called with `--statistics`.[1] For each variable, `llvm-dwarfdump`'s metric measures the length of the covered range in bytes relative to the range of the variable's scope. Thus, it depends on the characteristics of the generated machine code,

[1]See `https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html`, accessed 2024-08-29

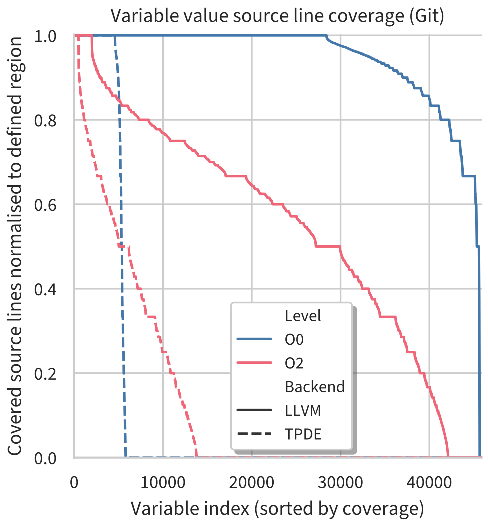


Figure 6.1: *Coverage results. For each compilation, the variables are sorted independently by their source line coverage.*

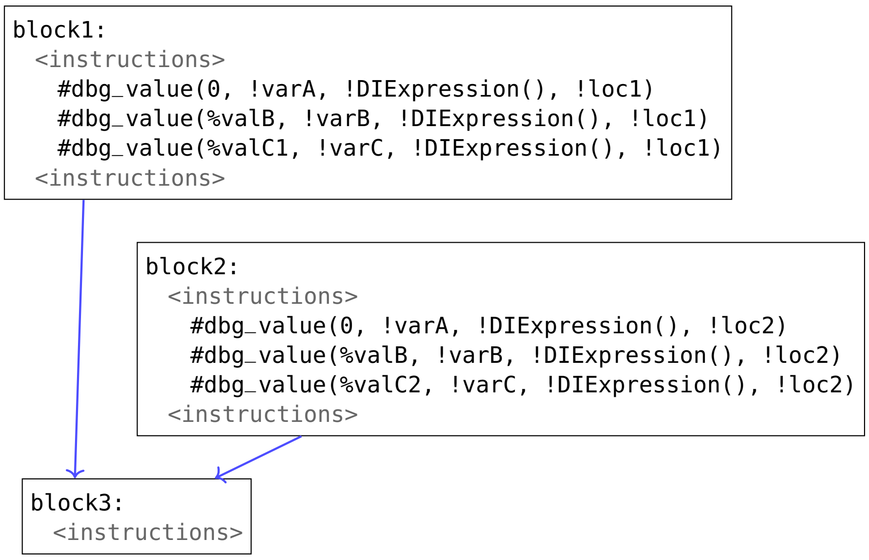
which makes it unsuitable for comparing different compilers. This is especially a problem when comparing LLVM and TPDE since the latter emits longer function prologues.

A metric that solves the issue of limited comparability between compilers was proposed in [17]. It counts *source lines* at which variables are described by debugging information and compares the result to the number of lines at which the variable has a *defined value* in the program. We applied this metric both to LLVM and TPDE. Like the authors of [17], we measured and visualized the local variable coverage for the *git* codebase[2] using their provided tools [18].

Figure 6.1 shows the results. While the differences in slope between `-O0` and `-O2` are similar for TPDE and LLVM, the curves corresponding to TPDE show that the major part of the variables is not covered at all. Because of the following observations, we assume that Figure 6.1 misses variables that are actually covered by TPDE's debugging information and we consider some kind of incompatibility between TPDE and the tools from [18] the reason of unreliable results.

- The program used to create Figure 6.1 emitted warnings about 25655 variable names at `-O0` and 17573 at `-O2` that are present in LLVM but not in TPDE.
- We picked three sample variables which were missing from the TPDE `-O0` curve according to intermediate files we produced during generation of Figure 6.1 and we manually inspected their locations via `llvm-dwarfdump`. All three variables had corresponding DIEs in the `.debug_info` section featuring non-empty location lists each of which covered the whole variable scope except for the function prologue.
- It is unlikely that TPDE can cover variables at `-O2` that it cannot describe at `-O0`.

[2]Commit bd5df96, Url: `https://git.kernel.org/pub/scm/git/git.git/commit/?id=bd5df96`, accessed 2024-08-29



```
block1:
  <instructions>
    #dbg_value(0, !varA, !DIExpression(), !loc1)
    #dbg_value(%valB, !varB, !DIExpression(), !loc1)
    #dbg_value(%valC1, !varC, !DIExpression(), !loc1)
  <instructions>
```

```
block2:
  <instructions>
    #dbg_value(0, !varA, !DIExpression(), !loc2)
    #dbg_value(%valB, !varB, !DIExpression(), !loc2)
    #dbg_value(%valC2, !varC, !DIExpression(), !loc2)
  <instructions>
```

```
block3:
  <instructions>
```

Figure 6.2: *Example in which the propagation simulation pass removes location information despite (possibly) identical locations in the predecessors.*

We expect the real coverage to be higher. While we do not know of particular reasons why TPDE would perform significantly worse than LLVM in the absence of `#dbg_value`, we anticipate a considerable loss of coverage when compiling optimized LLVM IR because of following reasons:

Aborted Expressions Due to the limited scope of this thesis, TPDE currently cannot handle all features of debug records. When it encounters unsupported constructs, it resorts to aborting the expression, which leads to a gap in coverage.

Value Lifetime Whenever the lifetime of an SSA value ends, TPDE terminates the ranges of all location list entries which depend on the value. However, values often remain at their last locations until these are reused. TPDE does not exploit this possibility to achieve higher coverage.

Pointer Comparison in Propagation Simulation When updating initial record configurations, the propagation simulation pass compares only the pointers of debug records. It thus also drops location information if the final record configurations of predecessor blocks contain different debug records providing the same value and expression. Two examples of this situation are shown in Figure 6.2: For both variables `!varA` and `!varB`, there are `#dbg_value` records in both predecessors, which assign the same value. While TPDE drops the variable locations at the beginning of `block3`, LLVM does not.

Additionally, the `LiveDebugValues` pass, which we can consider LLVM's counterpart of TPDE's propagation pass, is run after register allocation. Thus, it has access to the locations of variables in terms of physical registers and memory and can therefore even provide variable locations for the variable `!varC` in Figure 6.2 if the two values `%valC1` and `%valC2` share the same location. [11]

6.2 Compilation Time

In the following, we analyze the slowdown introduced by our changes to TPDE and compare it to the performance impact that debugging information has on LLVM's back-end.

6.2.1 Setup

We recorded the time Clang spent in the back-end for each compilation unit. For that we used Clang's `-ftime-trace` [1] functionality, which measures the execution time for various parts of the compiler. We note that this measurement introduces a small amount of overhead, which might be insignificantly higher for LLVM's back-end because it also records the timings of individual passes within the back-end.

We performed compilations for both mentioned optimization levels with and without the `-g` command line flag, which enables emission of debugging information. Because our modified version of TPDE can only be used in combination with the `-g` flag, we used the original TPDE version for compilations without the flag.

This resulted in eight different types of compilations per benchmark that we each repeated nine times. For each type of compilation and benchmark we removed the smallest and the largest compile time to compute a truncated arithmetic mean.

We obtained our results on a system with two Intel XEON E5-2690 v3 processors and 96 gigabytes of memory running Ubuntu 24.04 LTS. We did not execute multiple compilations in parallel and ensured that each process only allocated memory that belongs to the processor it was executed on.

6.2.2 Results and Discussion

In our results, which are depicted in Figure 6.3, the overhead of generating debugging information in TPDE for code at optimization level `-O0` ranges from 41% to 65% depending on the program. *deepsjeng* is an outlier that only features an overhead of 11%. On average, generating debugging information at `-O0` introduces an overhead of 45%.

The slowdowns at optimization level `-O0` are comparable to those of LLVM's back-end, which are also shown in Figure 6.3. In contrast, at `-O2`, the slowdown relative to compilations without debugging information is significantly higher in TPDE than in LLVM's back-end. On average, the time spent in the back-end nearly doubles.

We consider the following differences of LLVM IR generated by `clang -O2` relative to LLVM IR from `clang -O0` to be the primary causes of increased slowdown at `-O2`:

- **Reordered instructions:** Optimization passes can reorder instructions which link to `DILocation` nodes. This can lead to gaps in program counter ranges, resulting in longer range lists and additional lookups in hash maps.

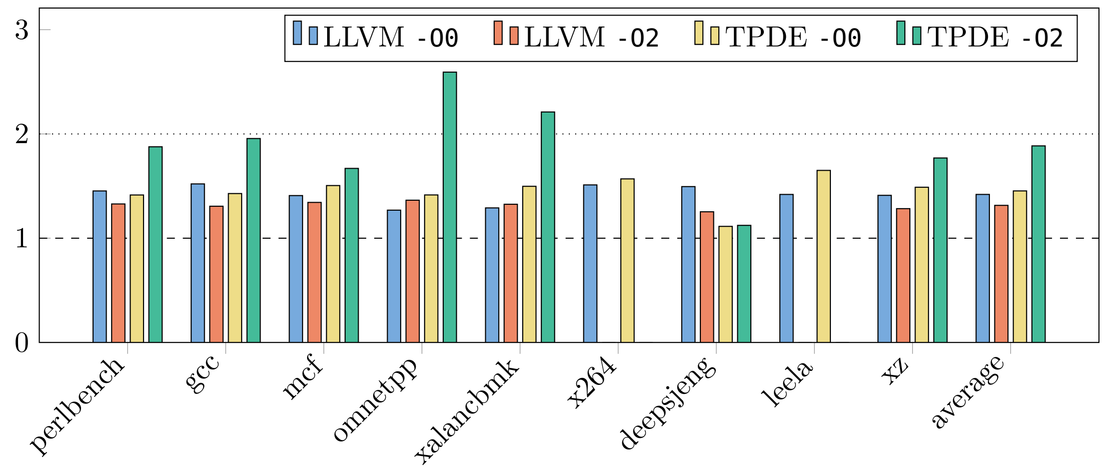


Figure 6.3: *Slowdown of the back-end when generating debugging information relative to corresponding compilations without debugging information.*

- **Inlinined functions:** At `-O2`, Clang performs inlining of functions. For each inlined instance of a function, an additional inlined instance tree has to be constructed and serialized.

- **Variables promoted to SSA values:** Near the beginning of the optimization pipeline, Clang executes the `mem2reg` pass, after which the locations of most variables are described by `#dbg_value` instead of `#dbg_declare`. Thus, the order of magnitude of debug records increases from one record per variable to one record per assignment. As a consequence, more location list entries have to be emitted. Our propagation simulation pass is also affected even though it does not distinguish between different kinds of debug records. While the complexity of the transfer function and the combining function does not depend on the number of debug records, the number of visits for each block is lower when there is only one debug record for each variable.

Reordered instructions and inlined functions also affect LLVM's back-end. The additional slowdown is, however, not visible in Figure 6.3 because code generation in LLVM also slows down when using `-O2`, whereas code generation in TPDE is slightly faster when `-O2` is used. The reason is that at `-O2`, due to our measuring setup, LLVM's back-end performs further target-specific optimizations, whereas TPDE does not. The relative compile time at `-O2` compared to `-O0` is shown in Figure 6.4.

In Figure 6.5, we show the speedup of TPDE relative to LLVM only at `-O0` for the aforementioned reason. We again notice an outlier, *deepsjeng*, which TPDE only compiles 1.6 times faster than LLVM for builds without debugging information. We consider this the reason for the reduced slowdown shown for *deepsjeng* in Figure 6.3.

On average, TPDE achieved a speedup of 7.1 relative to LLVM when debugging information was enabled. Compared to the average speedup of 7.6 we measured when compiling without debugging information, this indicates that not only the code generation of TPDE, but also our additions perform considerably faster than their counterparts

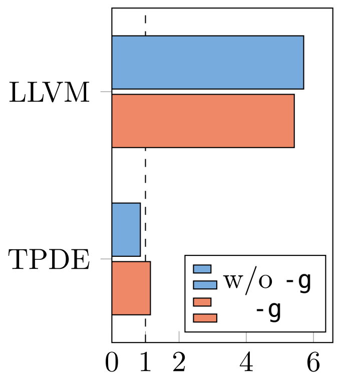


Figure 6.4: *Average slowdown of back-end relative to* `-O0` *introduced by* `-O2`

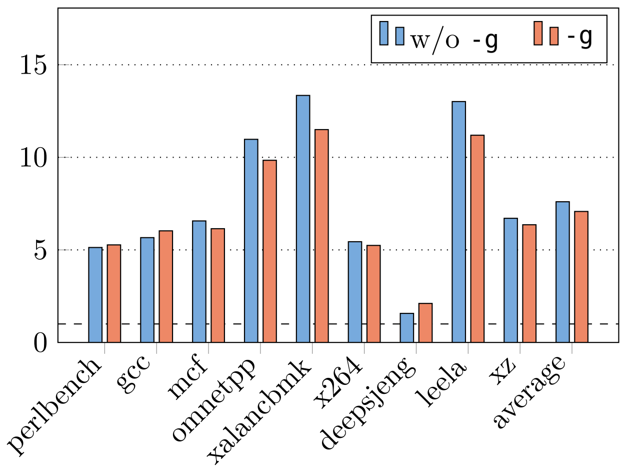


Figure 6.5: *Speedup of TPDE compared to LLVM at* `-O0`.

in LLVM. After inspecting parts of LLVM's sources, we assume that following factors substantially slow down LLVM's back-end relative to TPDE:

DIE Data Structure Similar to TPDE, the objects used in LLVM to represent DIEs prior to serialization are bump-allocated[3]. Unlike our intermediate nodes, which—since the fields are hardcoded—only store values, LLVM stores attributes with their forms and values as linked lists[4], which occupy more memory. This design is more flexible than that of TPDE, because it allows addition and removal of attributes at runtime.

Due to the small set of available abbreviations in TPDE, we were able to implement a mapping from intermediate nodes to matching abbreviations based on a array lookup. Due to the dynamic design of LLVM's data structure, an array based mapping is not applicable. LLVM therefore maintains a hash set containing unique abbreviations.[5] For each DIE, the set of attributes with forms as well as the tag and the flag specifying whether the DIE has children together serve as the key for lookup. Hashing a list is typically a costly operation.

Precomputation of Offsets Before LLVM serializes the DIEs, it computes their sizes to derive the offsets for references.[6] Thus, LLVM iterates at least twice through its tree of DIEs. In contrast, TPDE directly serializes DIEs in one pass and later patches forward references.

Machine IR Passes After instruction selection, LLVM executes further passes on Machine IR. Similar to debug records in LLVM IR, Machine IR features machine pseudo-instructions, which also appear interleaved with instructions. We can divide the back-end passes into

[3]Source: llvm/include/llvm/CodeGen/DIE.h:849
[4]Source: llvm/include/llvm/CodeGen/DIE.h:674
[5]Source: llvm/include/llvm/CodeGen/DIE.h:140
[6]Source: llvm/include/llvm/CodeGen/DIE.h:899

three categories: There are passes which are dedicated to debugging information, passes which read or modify debugging machine pseudo-instructions as a side effect and passes which do not interact with debugging information. Even the passes from the last category can be slowed down by debugging information since these passes have to skip machine pseudo-instructions when iterating over Machine IR. [11]

6.3 Section Sizes

To evaluate the space-efficiency of our chosen encodings, we compiled the benchmarks and extracted the section sizes from the fully-linked binaries. This measuring method has an impact on the results in the following ways: First, it only considers the DWARF sections themselves without their corresponding relocation sections, which are only present in relocatable object files. Second, the linker removes duplicate items in `.debug_str` and `.debug_line_str`.

Our results depicted in Figure 6.6 show only minor discrepancies in the `.debug_str` section sizes. Because of the deduplication performed by the linker, the smaller sections in TPDE's binaries indicate that named DIEs generated by LLVM were not emitted by TPDE. This meets our expectations as we do not translate all information found in LLVM IR to DWARF as outlined in Section 4.4.

In addition, also the sizes of the `.debug_line` sections are comparable. Nevertheless, TPDE produces DWARF debugging information at `-O0` that is on average 22% larger than that of LLVM. While this is primarily a result of the increased `.debug_info` section size, other differences largely balance each other out.

For example, the TPDE binaries do not feature the `.debug_str_offsets` and `.debug_addr` sections, which provide mechanisms that allow reusing relocated section offsets. Furthermore, there is an upper bound in TPDE for the maximum size of the `.debug_abbrev` section in object files, which results from the limited number of possible abbreviations. Because the linker concatenates the abbreviation tables of all object files, the size of `.debug_abbrev` in the linked binary mainly depends on the number of compilation units. In all our results, `.debug_abbrev` sections generated by TPDE are smaller than those from LLVM. Another visible difference is that TPDE also emits `.debug_loclists`, while LLVM does not. That is because of our choice, to encode locations for all variables in location lists even if the variables have a fixed position on the stack. Analogously, the `.debug_rnglists` sections generated by TPDE are also larger than those by LLVM because we chose to represent all ranges in range lists, no matter whether they are contiguous or not.

The increase in `.debug_info` section size results from our design prioritizing a low number of encoding variants for each payload type over space-efficient encoding. In particular, we preferred to emit an empty value where feasible instead of omitting an attribute. An example for this are `DW_AT_name` attributes providing empty strings whenever a name is not provided in LLVM IR. Also, we avoided LEB128-encoding where possible and do not let TPDE choose the sizes of integer encodings based on the values.

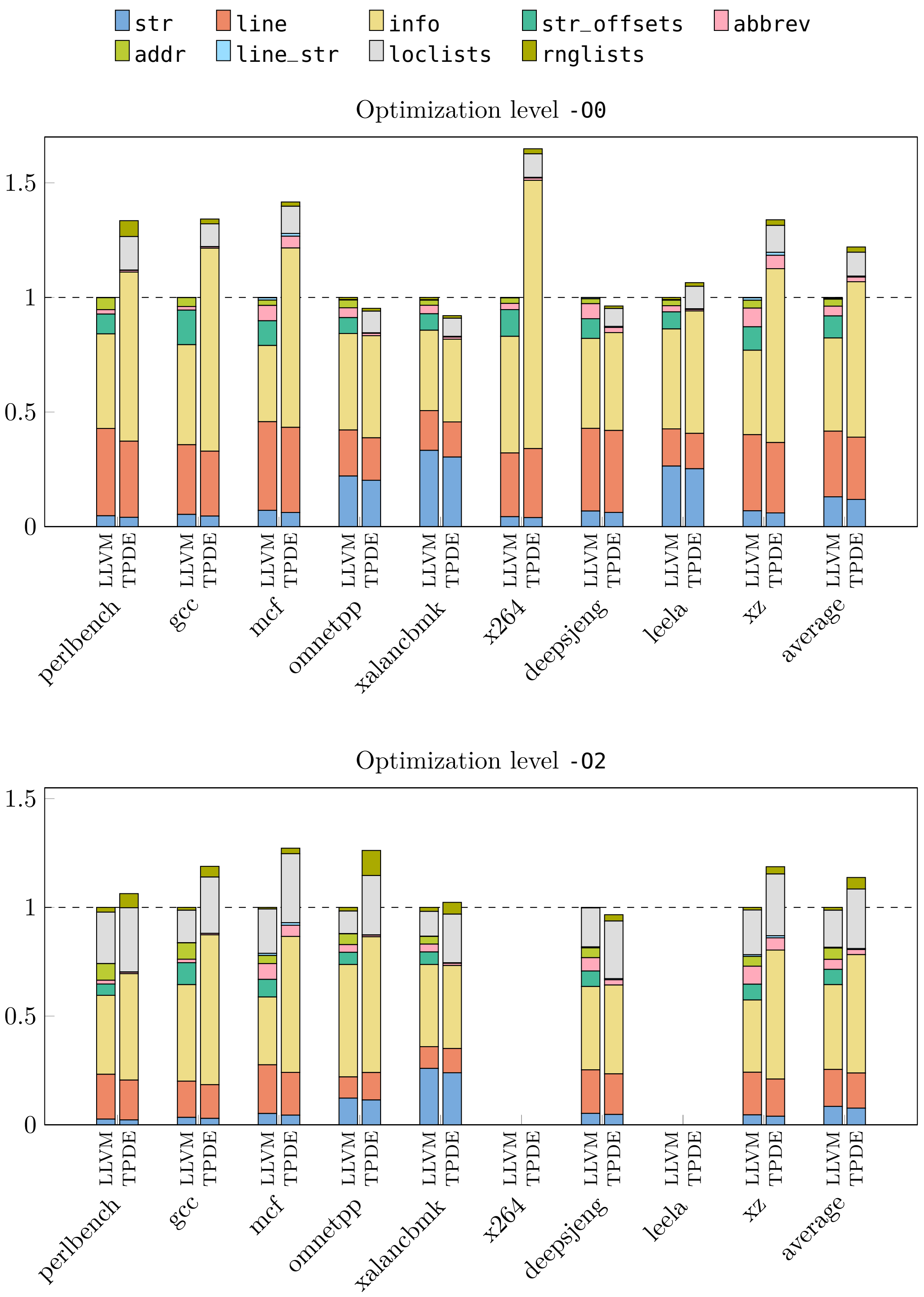


Figure 6.6: *Encoded size of DWARF information produced by TPDE relative to LLVM broken down to the different sections. The total size of debugging information produced by LLVM is normalized to 1 for each benchmark.*

7 Summary

The productivity of programmers can be increased by shortening compilation times. TPDE, an alternative LLVM compiler back-end, achieves this by reducing the number of passes from about 70 in typical compilers to 3. However, to be a viable tool for development, TPDE requires support for the emission of debugging information, which it was lacking prior to our work.

In this thesis, we implemented an emitter for DWARF version 5 debugging information that we integrated into TPDE. Our design focuses on a low impact on compilation time. Thus, we reused existing compiler passes for our purposes where possible. This in particular applies to the last pass, which in addition to instruction selection, register allocation, machine code emission and generation of stack unwinding information now also performs the synthesis of the line table and tracks further per-instruction information such scope ranges and variable locations. Although TPDE is primarily designed for the compilation of unoptimized code, our emitter can also handle features such as inlined functions and register-allocated variables that are more likely to be found in optimized LLVM IR. We evaluated our implementation by measuring the slowdown introduced by our emitter and we compared the size of the generated debugging information to that of LLVM.

Our additions to TPDE slow it down by 41%–65% when compiling unoptimized LLVM IR, which is a marginally higher slowdown than that which LLVM incurs when compiling with debugging information. At optimization level `-O0`, TPDE produces debugging information which is 22% larger than its LLVM counterpart. We also measured the variable coverage for both `-O0` and `-O2`, but the results showed inconsistencies.

Thus, we conclude that a reduced number of back-end passes not only speeds up code generation but also emission of debugging information. Our modified version of TPDE is largely usable, however further work is needed to evaluate the quality of its output quantitatively.

Acronyms

ABI	Application Binary Interface.
DIE	Debugging Information Entry.
LEB128	Little-Endian Base 128.
SSA	Static Single Assignment.

Bibliography

[1] Aras Pranckevičius. time-trace: timeline / flame chart profiler for Clang, January 2019. `https://aras-p.info/blog/2019/01/16/time-trace-timeline-flame-chart-profiler-for-Clang/`, Accessed: 2024-08-25.

[2] DWARF Debugging Information Format Committee. DWARF Debugging Information Format Version 5, February 2017. `https://dwarfstd.org/doc/DWARF5.pdf`, Accessed: 2024-08-27.

[3] Free Software Foundation. GCC, the GNU Compiler Collection, August 2024. `https://gcc.gnu.org`, Accessed: 2024-08-27.

[4] Free Software Foundation. GDB: The GNU Project Debugger, July 2024. `https://sourceware.org/gdb`, Accessed: 2024-08-27.

[5] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[6] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. AMD64 Architecture Processor Supplement, Version 1.0, June 2024. `https://web.archive.org/web/20240609215615/https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/7003569039/artifacts/raw/x86-64-ABI/abi.pdf`, Accessed: 2024-08-19.

[7] James Oakley. Exploiting the Hard-Working DWARF: Trojan and exploit techniques with no native executable code. In *5th USENIX Workshop on Offensive Technologies*, San Francisco, CA, August 2011. USENIX Association.

[8] LLVM Project. Clang: a C language family frontend for LLVM. `https://clang.llvm.org`, Accessed: 2024-08-27.

[9] LLVM Project. Debug Info Assignment Tracking. `https://llvm.org/docs/AssignmentTracking.html`, Accessed: 2024-08-23.

[10] LLVM Project. LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, Accessed: 2024-08-03.

[11] LLVM Project. Source Level Debugging with LLVM. `https://llvm.org/docs/SourceLevelDebugging.html`, Accessed: 2024-08-22.

[12] LLVM Project. The LLDB Debugger, August 2024. `https://lldb.llvm.org`, Accessed: 2024-08-27.

[13] LLVM Project. The LLVM Compiler Infrastructure, August 2024. `https://llvm.org`, Accessed: 2024-08-27.

[14] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[15] Tobias Schwarz and Alexis Engelke. Building a Fast Back-End for LLVM, March 2024. `https://llvm.org/devmtg/2024-03/slides/llvm-fast-backend.pdf`, Accessed: 2024-08-23.

[16] Standard Performance Evaluation Corporation. SPEC CPU 2017, December 2022. `https://www.spec.org/cpu2017/`, Accessed: 2024-08-24.

[17] J. Ryan Stinnett and Stephen Kell. Accurate Coverage Metrics for Compiler-Generated Debugging Information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 126–136, New York, NY, USA, 2024. Association for Computing Machinery.

[18] J. Ryan Stinnett and Stephen Kell. Accurate coverage metrics for compiler-generated debugging information (artifact), January 2024. `https://doi.org/10.5281/zenodo.10568392`.

[19] Keith Walker. A comparison of the DWARF debugging information produced by LLVM and GCC, November 2013. `https://llvm.org/devmtg/2013-11/slides/Walker-DWARF.pdf`, Accessed: 2024-08-17.