IMPLEMENTATION OF THE DEBUGGING SUPPORT FOR THE LLVM OUTLINING OPTIMIZATION

Vojislav Tomašević[1]*,
[0009-0000-5948-0123]
Đorđe Todorović[2],
[0009-0007-6109-8127]
Maja Vukasović[1]
[0000-0003-0647-1922]

[1]School of Electrical Engineering,
Belgrade, Serbia

[2]HTEC,
Belgrade, Serbia

Abstract:
Outlining optimization has been used in compilers predominantly to decrease the code size and sometimes even to improve its efficiency. If a code segment is repeated at various places in the code, the compiler can encapsulate it into a function and then it replaces these segments with the function calls. The *LLVM* infrastructure supports the outlining optimization but lacks proper debugging information in such cases and an outlined function cannot be differentiated from some other compiler-generated code at debug time. This paper proposes the complete solution to this problem on three levels of abstraction: *IR* and *Machine IR* code, *DWARF* format, and *LLDB* debugger. It identifies the reason for insufficient precision of the previous debugging information and describes in detail the implementation focused on enhancing these data so that the outlined function can be exactly recognized during debugging. The implementation has been thoroughly tested through regression and custom tests and it was made publicly available. In practice, the enhanced debugging information for outlining has proven to be useful.

Keywords:
Compilers, *LLVM* Infrastructure, Outlining Optimization, Debugging.

INTRODUCTION

In addition to its basic purpose of generating executable code, an important goal of a compiler is to make such code as efficient as possible, regarding both execution speed and memory usage. To this end, numerous optimizations are applied at various places to transform the code into its improved equivalent [1]. One of them is the outlining optimization supported by some compilers. Its main goal is to extract the same, replicated code from different places in the program into a new function and to replace all occurrences of the segment with a corresponding function call. In this way, the entire code size is decreased, and memory consumption is improved.

The compilers need to overcome the high abstraction level and produce efficient code, so they consist of many components. The compilation runs in certain consecutive phases that sometimes partly overlap. Thus, the compilers are quite complex and hard to develop. Because of that, it is very important to have some suitable infrastructure platforms

Correspondence:
Vojislav Tomašević

e-mail:
vojkan99@gmail.com

DOI: 10.15308/Sinteza-2025-233-240

that make creating the compilers for arbitrary languages and target machines less demanding. They provide ready-made tools and projects that can be used and customized for the development of a proprietary compiler. One of the best-known and widely used platforms is the *LLVM* infrastructure [2]. Besides the tools used in the compilation process, e.g. *Clang* front-end compiler for *C* and *C++* languages, *LLVM* contains rich support for debugging providing a vast amount of data about the program, stored in a convenient form.

LLVM does support the outlining optimization pass, but there is a problem related to information relevant to the debugging process since the outlined code cannot be differentiated from the code generated in some other way. To allow better precision in the interpretation of the code and to enable more efficient debugging, it is necessary to enhance the *LLVM* infrastructure with some additional information that indicates the occurrence and location of the outlined code. This paper proposes a solution to the previous problem by expanding debugging-related information in the context of outlining optimization. The proposal is successfully implemented and thoroughly tested using *LLVM* and *LLDB* testing infrastructures.

The second section of this paper explains the outlining optimization itself, its benefits, and how it is performed. The third section focuses on the solution proposal and describes its implementation on all three levels in detail. The implementation is illustrated with the appropriate code segments. The conclusion summarizes the paper.

2. OUTLINING OPTIMIZATION

Outlining is an optional part of the compilation process that removes a part of the code from a function and replaces it with the call of the new function, which consists of the removed code [3]. The new function is artificially generated by the compiler (more precisely, by its optimizer), and we say that the function is outlined. This process is the opposite of the more frequently employed inlining optimization that embeds the body of a function at its call site, to speed up the program by eliminating the function-call overhead. Since the same function can be inlined at multiple call sites, if not applied selectively, this optimization may cause severe code growth [4].

On the other hand, outlining results in additional calls of the newly created function, which can slow down the execution. However, it pays off by decreasing the code size if the same outlined code is replicated multiple times. Therefore, outlining optimization is useful predominantly in systems where memory is a critical resource, but which are also fast enough so that the additional function calls don't result in significant performance degradation (e.g., microcontrollers).

Typical applications of outlining are: code refactoring or extraction of the kernel in compilers which transform the source code written in one language to another [5], and shortening of large functions to decrease the compilation time in *JIT* compilers [6]. Another source of potential performance gain can be obtained in cases when a large, frequently called function (hot function) contains some rarely executed regions (cold code) [3]. Outlining of cold code sections from a hot function can have at least three advantages:

1. Removing cold code from a large hot function can make it small enough to apply inlining;
2. Outlining the cold code can improve the cache memory efficiency by preserving the spatial locality of hot code;
3. Outlining the cold code from a hot function can also improve memory bandwidth during instruction fetching, which is important for modern superscalar and *VLIW* architectures.

Outlining optimization in the *LLVM* infrastructure can be carried out on both *IR* and *MIR* code levels. The algorithm of outlining on the *MIR* level consists of the following phases [6]:

1. *Identification of the candidates* – In this initial phase, all basic blocks [1] in the program are searched for the longest repeated sequence of instructions. This resembles the problem of finding the longest common substring where basic blocks act as strings and instructions act as characters. This problem can be solved by using the suffix tree program representation;
2. *Removing unsafe or useless candidates* – After the candidates are found, the potential adverse effects of their outlining should be examined (e.g., some instructions like conditional branches cannot be safely extracted from the function). Thus, unsafe candidates are rejected as well as those candidates that do not contribute to the decrease of the code size. The formula for calculating the usefulness of the outlining process is given in [6];
3. *Function sharing* – After the list of candidates is finalized, code transformation takes place. The new functions are created, and each candidate is replaced with the corresponding calls.

3. IMPLEMENTATION

Although the *LLVM* infrastructure supports the outlining optimization, its implementation is not complete enough as far as the debugging information is concerned. As for the debugging data, *LLVM* treats the new functions generated in case of outlining the same way as any other compiler-generated code. This shortcoming motivated us to propose a solution within the *LLVM* that makes a distinction between outlined code and other compiler-generated code [7]. This solution required some modifications on three levels of abstraction: *IR* and *MIR* code, *DWARF* format, and *LLDB* debugger. This proposal covers a wide range of the *LLVM* debug information that is useful for both the user and programmer to see whether the outlining is applied somewhere in the code. Support for these three levels is presented in the following text. The entire implementation is publicly available [8].

3.1. THE *IR* AND *MIR* CODE LEVELS

IR (*intermediate representation*) code used in *LLVM* is generated by the front-end and the middle-end (i.e., the optimizer) of the compiler [9]. Its advantages are the architectural independence and separation between the compiler front-end and back-end, which enables the different implementations of these two parts to connect and work together. This kind of intermediate code is very suitable for optimizations. *MIR* (*machine-specific intermediate representation*) is also a type of intermediate code that is used in *LLVM* in the compiler back-end. It is generated after the instruction selection phase of the compilation. *MIR* code is also very convenient for target architecture-aware optimizations that are performed in the back-end. When debugging information is enabled, *IR* code keeps it in the form of the *LLVM* metadata. Since *MIR* is an extension of *IR* and each *MIR* module contains a corresponding *IR* module, *MIR* refers to the same metadata with debug information from the contained *IR* module. All metadata in the *LLVM* infrastructure represents class objects derived from the *llvm::Metadata* base class [10].

Outlining optimization in the *LLVM* infrastructure is implemented on both *IR* and *MIR* levels at the module level in their *runOnModule* functions. Besides the generation of an outlined function, its corresponding metadata node *DISubprogram* is also created by calling *createFunction* of the *DIBuilder* class. For this node, there is the *DINode::DIFlags::FlagArtificial* flag that is set to indicate that the function is artificially created by the compiler. Listing 1 shows a part of the *IR* file for the sample program after outlining two functions *outlined_ir_func_0* and *outlined_ir_func_1*. The *DISubprogram* metadata nodes for them contain *DIFlagArtificial* indicator which informs that these functions are artificially generated by the compiler. However, it is not clear that they are outlined.

```
; Function Attrs: nounwind uwtable
define dso_local i32 @main() #0 !dbg !7 {
entry:
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  call void @outlined_ir_func_1(i32* %x), !dbg !14
  call void @outlined_ir_func_1(i32* %y), !dbg !14
  call void @outlined_ir_func_0(i32* %x, i32* %y), !dbg !14
  call void @outlined_ir_func_0(i32* %y, i32* %x), !dbg !14
  ret i32 0
}

!15 = distinct !DISubprogram(name: "outlined_ir_func_0", linkageName: "outlined_ir_func_0",
scope: !1, file: !1, type: !16, flags: DIFlagArtificial, spFlags: DISPFlagDefinition |
DISPFlagOptimized, unit: !0, retainedNodes: !17)
!16 = !DISubroutineType(types: !17)
!17 = !{}
!18 = !DILocation(line: 0, scope: !15)
!19 = distinct !DISubprogram(name: "outlined_ir_func_1", linkageName: "outlined_ir_func_1",
scope: !1, file: !1, type: !16, flags: DIFlagArtificial, spFlags: DISPFlagDefinition |
DISPFlagOptimized, unit: !0, retainedNodes: !17)
```

Listing 1. A part of the resulting *IR* file after the outlining is applied to a sample program [7]

The solution to this problem is found in the addition of a new flag named *Outlined* in the *DIFlags enum* statement. This *enum* statement contains all flags for some metadata and it is a part of the *DINode* class also inherited by the *DISubprogram* class. The role of the new flag is to reflect the information that the corresponding *DISubprogram* is an outlined function. Like all other flags, the *Outlined* flag is declared in *llvm/include/llvm/IR/DebugInfoFlags.def* file, whose content is included in the *enum* statement. The flag was then added by creating an appropriate call of the *HANDLE_DI_FLAG* macro, which has previously been defined in the *DIFlags enum* statement. After that, the definition of the *Largest* flag was also updated accordingly. These code modifications are shown in Figure 1.

After adding the new flag, its use on both *IR* and *MIR* levels was implemented. When an outlined function and its corresponding *DISubprogram* object are created, instead of a more general *DINode::DIFlags::FlagArtificial* indicator which tells that the function does not exist in the source code, the new *DINode::DIFlags::FlagOutlined* flag is set. It provides more precise information about the nature of the compiler-generated function, as illustrated in Figure 2. Modification on the *MIR* level was performed in a quite similar manner.

Now, after the outlining is applied, debugging information in the previous example is shown in Listing 2.

3.2. DWARF FORMAT LEVEL

The *DWARF* format is one of the most frequently used formats for debugging information [11]. According to *DWARF*, debug information is represented as a tree-like structure. Hence, it consists of the *DIE* nodes connected by basic parent-children relationships. The characteristics of program entities represented by the *DIE* objects are described by a versatile set of attributes of different types (can even be a reference to some other DIE node).

One of the *DIE* objects is also the *DW_TAG_subprogram* object which represents the function. Among other attributes, it contains the *DW_AT_artificial* attribute whose purpose is to denote the compiler-generated construct that does not exist in the source code. Again, this information is not precise enough in the case of outlined functions. Therefore, the outlined code needs to be distinguished from some other artificially generated code.

3 llvm/include/llvm/IR/DebugInfoFlags.def

```
@@ -58,6 +58,7 @@ HANDLE_DI_FLAG((1 << 26), NonTrivial)
58  58  HANDLE_DI_FLAG((1 << 27), BigEndian)
59  59  HANDLE_DI_FLAG((1 << 28), LittleEndian)
60  60  HANDLE_DI_FLAG((1 << 29), AllCallsDescribed)
    61  + HANDLE_DI_FLAG((1 << 30), Outlined)
61  62
62  63  // To avoid needing a dedicated value for IndirectVirtualBase, we use
63  64  // the bitwise or of Virtual and FwdDecl, which does not otherwise
@@ -67,7 +68,7 @@ HANDLE_DI_FLAG((1 << 2) | (1 << 5), IndirectVirtualBase)
67  68  #ifdef DI_FLAG_LARGEST_NEEDED
68  69  // intended to be used with ADT/BitmaskEnum.h
69  70  // NOTE: always must be equal to largest flag, check this when adding new flag
70      - HANDLE_DI_FLAG((1 << 29), Largest)
    71  + HANDLE_DI_FLAG((1 << 30), Largest)
71  72  #undef DI_FLAG_LARGEST_NEEDED
72  73  #endif
73  74
```

Figure 1. Introducing the new *Outlined* flag [8]

2 llvm/lib/Transforms/IPO/IROutliner.cpp

```
@@ -624,7 +624,7 @@ Function *IROutliner::createFunction(Module &M, OutlinableGroup &Group,
624 624         0 /* Line 0 is reserved for compiler-generated code. */,
625 625         DB.createSubroutineType(DB.getOrCreateTypeArray(None)), /* void type */
626 626         0, /* Line 0 is reserved for compiler-generated code. */
627     -       DINode::DIFlags::FlagArtificial /* Compiler-generated code. */,
    627 +       DINode::DIFlags::FlagOutlined /* Compiler-generated outlined code. */,
628 628         /* Outlined code is optimized code by definition. */
629 629         DISubprogram::SPFlagDefinition | DISubprogram::SPFlagOptimized);
630 630
```

Figure 2. Using the Outlined flag on the *IR* level [8]

```
!15 = distinct !DISubprogram(name: "outlined_ir_func_0", linkageName: "outlined_ir_func_0",
scope: !1, file: !1, type: !16, flags: DIFlagOutlined, spFlags: DISPFlagDefinition |
DISPFlagOptimized, unit: !0, retainedNodes: !17)
!16 = !DISubroutineType(types: !17)
!17 = !{}
!18 = !DILocation(line: 0, scope: !15)
!19 = distinct !DISubprogram(name: "outlined_ir_func_1", linkageName: "outlined_ir_func_1",
scope: !1, file: !1, type: !16, flags: DIFlagOutlined, spFlags: DISPFlagDefinition |
DISPFlagOptimized, unit: !0, retainedNodes: !17)
```

Listing 2. *DISubprogram* metadata of the outlined functions from Listing 1 after *Outlined* flag is introduced [7]

Figure 3. Declaration of the *DW_AT_LLVM_outlined DWARF* attribute [8]

Figure 4. Adding the *DW_AT_LLVM_outlined* attribute to the *DIE* object of the function [8]

Translation of a program in *LLVM* acts as a pipeline. It enables the propagation of information generated in an earlier phase of compilation to the later stages. This property was exploited in augmenting the debugging information regarding outlining optimization in the *DWARF* format.

Since the new, *Outlined* indicator is added on the *IR* level, it is propagated through the compiler up to the place where the support for building the debugging information in *DWARF* format is implemented as an indication that some function is outlined. It is achieved by introducing the *isOutlined* function in the *DISubprogram* class which checks whether the *Outlined* indicator for a given object is set. If the call of the *isOutlined* function returns a *true* value, the attribute *DW_AT_LLVM_outlined* is added to the corresponding *DIE* object representing the outlined function. This attribute is introduced in the *DWARF* format by adding another call of the *HANDLE_DW_AT* macro with the *LLVM_outlined* argument. It was done in the part reserved for the extension of the format for the *LLVM* project needs, in the file *llvm/include/llvm/BinaryFormat/Dwarf.def* file as shown in Figure 3.

The addition of this attribute in the *DIE* object related to the *DISubprogram* metadata is shown in Figure 4. Listing 3 shows the contents of the *DW_TAG_subprogram* objects of the two outlined functions from Listing 1 after the proposed modifications are implemented.

```
0x0000005b:     DW_TAG_subprogram
                  DW_AT_low_pc          (0x000000000000004e)
                  DW_AT_high_pc         (0x0000000000000052)
                  DW_AT_frame_base              (DW_OP_reg7 RSP)
                  DW_AT_linkage_name            ("outlined_ir_func_1")
                  DW_AT_name            ("outlined_ir_func_1")
                  DW_AT_LLVM_outlined           (true)
                  DW_AT_external        (true)

0x00000065:     DW_TAG_subprogram
                  DW_AT_low_pc           (0x000000000000003c)
                  DW_AT_high_pc          (0x000000000000004e)
                  DW_AT_frame_base              (DW_OP_reg7 RSP)
                  DW_AT_linkage_name            ("outlined_ir_func_0")
                  DW_AT_name            ("outlined_ir_func_0")
                  DW_AT_LLVM_outlined           (true)
                  DW_AT_external        (true)
```

Listing 3. New display of the *DW_TAG_subprogram* objects for two outlined functions [7]

3.3. *LLDB* DEBUGGER LEVEL

LLDB debugger is also developed as a part of the *LLVM* project [12]. Like every other debugging tool, it accepts executable code as input and then executes it using the debugging information, with the provided parameters. During the debugging process, in some situations, *LLDB* prints messages to the user that help him understand the program being executed, as well as the modifications of the program resulting from applied optimizations. Enhancing this information regarding outlining optimization in the *LLDB* debugger is the main goal at this level.

The problem is quite like to the problems we have faced on the *IR/MIR* and *DWARF* levels. If a function is outlined, the *LLDB* tool recognizes it as artificially generated by the compiler, without any further explanation. Namely, during the execution of an outlined function, the *LLDB* debugger prints the following message „*Note: this address is compiler-generated code that has no source code associated with it.*“ This is a generic message which is printed each time some compiler-generated code is encountered. If the name of the function with the compiler-generated code is known, it is also given in the previous message as a parameter. Whether some code is compiler-generated or not is checked by the *GetStatus* method of the *StackFrame* class, which prints the description of the stack frame and/or source (or assembly) code context for this frame. This method examines whether there is a source code line that corresponds to the current address during debugging. If the *line* field of the *line_entry* object of the *LineEntry* class that represents a field of the *m_sc* object of the *SymbolContext* class is equal to zero, it is an indication of an artificially generated code.

To solve the previous ambiguity, it is designed that the *LLDB* debugger prints a new, custom message when an outlined function is encountered. Unlike previous messages about compiler-generated code that appear only once, the new message is printed after each instruction of an outlined function is executed, provided that step-by-step debugging mode is active.

Implementation of this *LLDB* debugger extension is based on the Function class since it represents a connection between the *DWARF* format and the *LLDB* as far as the function debugging is concerned. It contains information on whether the function represented by the object of this class is outlined. For this purpose, a new *m_outlined* field is added as well as the corresponding parameter of the class constructor for the initialization of the new field (the default value is *false*). Also, this class is extended with the *IsOutlined* method that returns the value of the *m_outlined* field.

Finally, it was necessary to implement the support for setting the new field in the following way. By parsing the die *DWARFDIE* object, the *ParseFunctionFromDWARF* method of the *DWARFASTParserClang* class creates the object of the Function class and returns the pointer to it. This *die* must have the *DW_TAG_subprogram* tag, so that it represents a function. During the parsing of the object, a *for* loop that traverses all *DWARF* attributes of the object is added and, if the new *DW_AT_LLVM_outlined* attribute is found, it keeps the record that the function is outlined by setting a local variable to *true*. This variable is later forwarded to the Function class constructor as a parameter that initializes the *m_outlined* field.

In this way, the information of the applied outlining optimization is propagated from the *DWARF* format to the *LLDB* tool. Previous implementation details are presented in Figure 5.

After the object of the *Function* class is created, it can be used to detect whether some function is outlined or not. This possibility is exploited in the *GetStatus* method of the *StackFrame* class. Besides reporting two previous messages about the existence of the artificially generated code, the new message „*Note: this function is outlined.*“ is displayed in case the *IsOutlined* method for the current function returns value *true*. This change is illustrated in Listing 4.

4. CONCLUSION

The debugging process is crucial for software testing and its efficiency essentially depends on the completeness and the precision of the debugging information. In the *LLVM* project, debug data cannot recognize the situation when a function is compiler-generated by outlining. This paper proposes a solution to this problem by enhancing the debugging information and its appropriate handling. The proposal is carefully implemented on the three levels of abstraction: the *IR* and *MIR* code, *DWARF* format, and *LLDB* debugger. Thorough regression testing was carried out to verify the correctness of the solution. It required enhancing the existing tests and writing new ones.

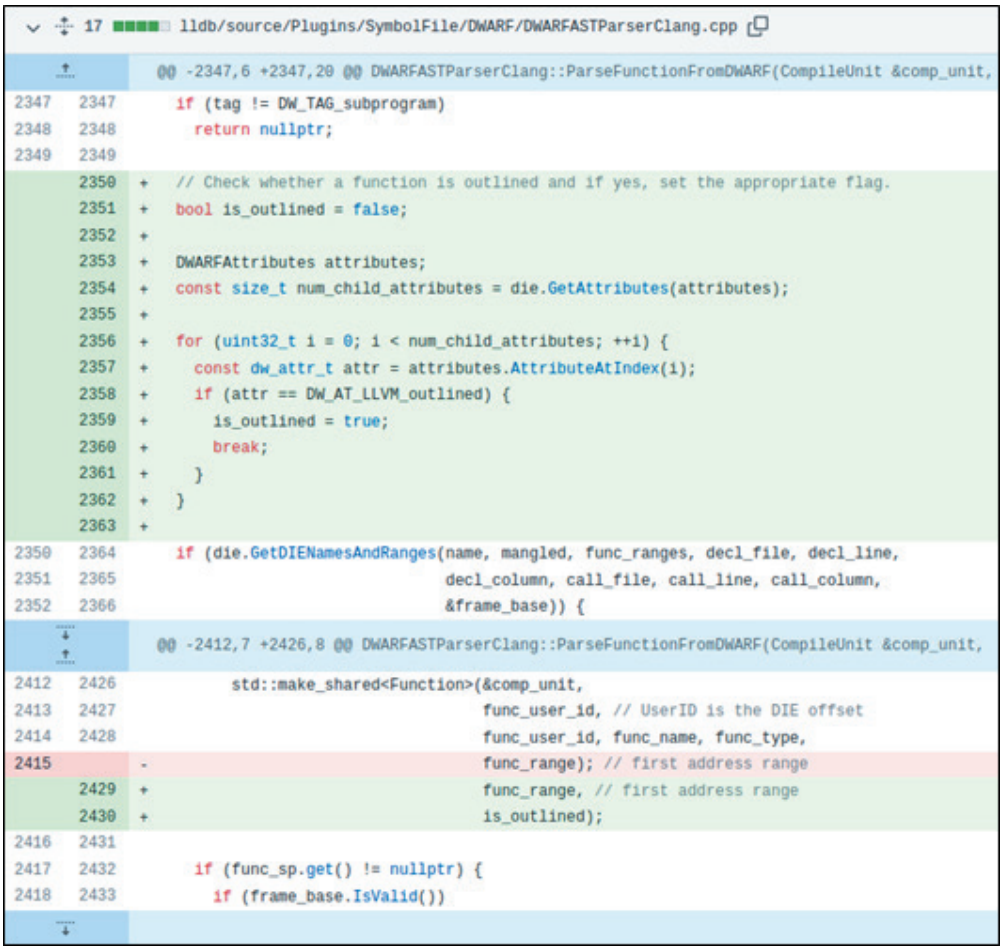

Figure 5. Propagation of information about outlining from *DWARF* to *LLDB* [8]

```
if (m_sc.function && m_sc.function->IsOutlined()) {
  strm.Printf("Note: this function is outlined.");
  strm.EOL();
}
```

Listing 4. The new message for an outlined function in the *GetStatus* method of the *StackFrame* class [7]

The implementation of the support for enhancing the debugging information in the context of the outlining optimization does not require the writing of a large amount of code, as can be seen from the proposed solution. Instead, it required a very demanding analysis of a complex project like *LLVM* and inserting many small changes at different places in the code to encompass all levels of abstraction in the *LLVM* project followed by exhaustive testing. Hence, the main part of the task is of the research type, while the implementation is a less demanding part. It is a typical pattern in the *LLVM* infrastructure when the modification or addition of a relatively small amount of code can provide a significant effect. There is also an idea for further improvement of user experience during debugging in *LLDB*. It can be achieved by providing support for the reconstruction of local variables in the outlined functions.

REFERENCES

[1] M. L. Scott, Programming Language Pragmatics, Morgan Kauffman, 2006.

[2] "The LLVM Compiler Infrastructure," [Online]. Available: https://llvm.org/. [Accessed 27 March 2025].

[3] P. Zhao and J. N. Amaral, "Function Outlining," Dept. of Computing Sciences, Univ. of Alberta, Edmonton, Canada, 2010.

[4] M. Vukasović and A. Prokopec, "Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code," *ACM Transactions on Programming Languages and Systems*, vol. 45, no. 4, pp. 1-64, 2023.

[5] C. Liao, D. J. Quinlan, R. Vuduc and T. Panas, "Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization," in *International Workshop on Languages and Compilers for Parallel Computing*, Newark, DE, United States, 2009.

[6] "Reducing code size with LLVM Machine Outliner on 32-bit Arm targets," [Online]. Available: https://www.linaro.org/blog/reducing-code-size-with-llvm-machine-outliner-on-32-bit-arm-targets/. [Accessed 30 August 2022].

[7] V. M. Tomašević, "Unapređenje infrastrukture LLVM dodavanjem informacija za otklanjanje grešaka prilikom autlajning optimizacije," School of Electrical Engineering, University of Belgrade, 2022.

[8] "[Outliner] Add debug-info support in IR, DWARF and LLDB," [Online]. Available: https://github.com/llvm/llvm-project/commit/80e1c808dd121595f7124917dd7ef22bb0da5fa7?diff=unified. [Accessed 25 March 2025].

[9] "Opt – LLVM optimizer," [Online]. Available: https://llvm.org/docs/CommandGuide/opt.html. [Accessed 20 March 2025].

[10] "llvm::Metadata Class Reference," [Online]. Available: https://llvm.org/doxygen/classllvm_1_1Metadata.html. [Accessed 20 March 2025].

[11] "DWARF Debugging Information Format Version 5," [Online]. Available: https://dwarfstd.org/doc/DWARF5.pdf. [Accessed 17 March 2025].

[12] "The LLDB Debugger," [Online]. Available: https://lldb.llvm.org/. [Accessed 23 March 2025].