SPLENDID: Supporting Parallel LLVM-IR Enhanced Natural Decompilation for Interactive Development

Zujun Tan
Princeton University, USA

Yebin Chon
Princeton University, USA

Michael Kruse
Argonne National Lab, USA

Johannes Doerfert
Argonne National Lab, USA

Ziyang Xu
Princeton University, USA

Brian Homerding
Northwestern University, USA
Argonne National Lab, USA

Simone Campanoni
Northwestern University, USA

David I. August
Princeton University, USA

ABSTRACT

Manually writing parallel programs is difficult and error-prone. Automatic parallelization could address this issue, but profitability can be limited by not having facts known only to programmers. A parallelizing compiler that collaborates with the programmer can increase the coverage and performance of parallelization while reducing the errors and overhead associated with manual parallelization. Unlike collaboration involving analysis tools that report program properties or make parallelization suggestions to programmers, decompiler-based collaboration could leverage the strength of existing parallelizing compilers to provide programmers with a natural compiler-parallelized starting point for further parallelization or refinement. Despite this potential, existing decompilers fail to achieve this goal because they do not generate portable parallel source code compatible with any compiler of the source language. This paper presents SPLENDID, an LLVM-IR to C/OpenMP decompiler that enables collaborative parallelization by producing standard parallel OpenMP code. Using published manual parallelization of the PolyBench benchmark suite as a reference, SPLENDID's collaborative approach produces programs twice as fast as either Polly-based automatic parallelization or manual parallelization alone. SPLENDID's portable parallel code is also more natural than what existing decompilers generate, obtaining a 39x higher average BLEU score.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Source code generation**.

KEYWORDS

decompilation, collaborative parallelization

ACM Reference Format:
Zujun Tan, Yebin Chon, Michael Kruse, Johannes Doerfert, Ziyang Xu, Brian Homerding, Simone Campanoni, and David I. August. 2023. SPLENDID: Supporting Parallel LLVM-IR Enhanced Natural Decompilation for Interactive Development. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582058

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582058

1 INTRODUCTION

The demand for performance and efficiency drives research to find better program parallelization methods. While recent breakthroughs [3, 4, 37] demonstrate the promising future of automatic parallelization, programmers will always play a role in parallelization. This is because, unlike a programmer, the compiler cannot expand the set of valid outputs of a program, even if such additional outputs would produce much better performance and be accepted as valid by the programmer [6, 48, 50]. For example, while the programmer may find the out-of-order printing of diagnostic messages acceptable for some level of performance, the compiler does not know this and cannot unilaterally make this change [48]. Additionally, the programmer may find lower precision for floating-point operations acceptable. The compiler, however, cannot relax the precision of output without the programmer playing a role. For this fundamental reason and other practical reasons, such as the current limits of parallelizing compilers, parallelization will always benefit from the programmer and compiler working together (collaborative parallelization).

Most parallelizations are, to some degree, a collaboration between the programmer and a compiler. First, the programmer can parallelize the program using a parallel programming language [8, 42, 57], parallel extensions to sequential languages [18, 40, 44], or by expressing code properties that enable inherent parallelism (i.e., implicit parallel programming [6, 7, 23, 25, 48, 61]). Then, the compiler maps this programmer-expressed parallelism to utilize parallel resources of the hardware. However, the degree of collaboration is limited in this way, either because the compiler performs only a translation of programmer-expressed parallelism or because the compiler disregards the work of the programmer and parallelizes the code itself (e.g., Polly [21]). In either case, only the programmer or the compiler is ultimately responsible for the parallelization choices. Alternatively, the compiler can collaborate with the programmer in parallelization by first presenting its parallelization to the programmer. It can do so in various ways. First, the compiler can present to the programmer assembly code, but this is too

obfuscated to find parallelism. Alternatively, many analysis tools assist manual parallelization by suggesting actions desired from the programmer (e.g., dependences to remove [24, 31, 62]). Unfortunately, collaboration based on analysis tools does not reduce the work of a programmer because the programmer must consider and make each change suggested by the tool. Moreover, requesting improvements for only some code regions limits the impact of the programmer since the compiler assumes that other regions can no longer be manually improved.

Another way to present compiler parallelization to a programmer is through decompilation. Decompilers [1, 10, 12, 14, 17, 22, 33, 53, 63, 64] have great potential to enable collaboration in which better performance can be obtained with less manual effort. Better performance can be achieved by making the work of one parallelizing compiler recompilable universally using any host compiler (portability). Ideally, to practically reduce manual effort, the style of the decompiled code should comply with contemporary coding practices, such as using a modern parallel programming model (naturalness). Natural code is informative about what and how a compiler parallelizes, enabling the programmer to improve program performance in any desired workflow.

When it comes to parallel programs, state-of-the-art decompilers cannot produce portable code. Translating parallel IR to portable parallel source code is not a trivial task. First, most parallel programming models impose strict requirements for loop structures. For example, the OpenMP [44] *omp for* construct requires syntactically canonical *for* loops with no additional code between the pragma and the loop. However, most decompilers end up translating low-level parallelized loops into *do-while* loops. This is because parallelization often relies on loop rotation for canonicalization, which converts all loops into *do-while* form. Furthermore, parallelism in the IR is often expressed using parallel runtime library calls. For reverse engineering purposes, code decompiled by previous decompilers exposes these library calls, making the decompiled code not recompilable with compilers using another runtime library.

Additionally, since code produced by state-of-the-art decompilers is not portable, it is also not natural. A *do-while* loop compared with a *for* loop is less natural without features like induction variables. Low-level runtime-specific details of parallelization also obfuscate previously decompiled code. While making the decompiled code portable helps with naturalness, code decompiled in previous approaches assigns variables with names corresponding to physical registers. The lack of informative variable names intrudes significant overhead in understanding code semantics.

To overcome the obstacles mentioned above and practically enable collaborative parallelization, this work proposes SPLENDID, the first LLVM-to-C/OpenMP decompiler that provides portable natural translation from parallel LLVM-IR to OpenMP-parallel source code. SPLENDID explicitly represents parallelism through the widely used parallel programming model, OpenMP [44]. Since using OpenMP directives eliminates compiler-specific implementations of parallel constructs and requires *for*-loops, SPLENDID-produced parallel code is portable and more natural. Moreover, code generated by SPLENDID preserves variable names and is thus closer to manually written code. With variable names that are representative of semantics, SPLENDID significantly reduces the manual effort of interpreting code semantics.

SPLENDID is designed with the careful consideration of what to de-transform so that key optimizations, such as parallelizations and loop optimizations, are made evident to the programmer. While the goal of this paper is to enable better collaborative parallelization, readers may find the results useful to other tasks that may benefit from more natural reverse engineering, such as debugging. For example, SPLENDID may be used in debugging and performance tuning of computational kernels automatically parallelized using Polly [21], an LLVM-based parallelizing compiler.
The primary contributions of this paper are:

- Presenting the first decompiler targeting OpenMP-parallel IR, SPLENDID. SPLENDID-produced code makes the output of a parallelizing compiler **portable**, recompilable with any host compiler, and **natural** for easy programmer involvement.
- A novel pass that **restores source variable names** by eliminating virtual register to variable naming conflicts and by inferring variable names from another function through inlining.
- Realizing a smarter **trade-off** between how close the decompiled code is to the original source code and how instructive it is to compiler parallelization.
- When SPLENDID-decompiled code is recompiled using GCC, an average speedup of 11x of 16 PolyBench benchmarks is made available universally outside LLVM. The same benchmarks demonstrate an average of 39x improvement on the BLEU score (i.e., a widely-used naturalness metric [45]) over the best prior work. With an average of 3 lines of manual change on top of SPLENDID-generated code, the speedup is doubled relative to both manual and compiler parallelization alone on 7 PolyBench benchmarks, programs simple enough that either the compiler or the programmer should have easily been able to deliver maximal performance but did not.

2 MOTIVATION

Decompilation has great potential to lower the manual effort of parallelizing programs. Portions of what otherwise be manual parallelization can be replaced by compiler parallelization if the decompiled parallel code is portable. The decompiled parallel code can then be maintained in place of the original sequential code. After seeing what the compiler parallelizes in the decompiled source code, the programmer can focus primarily on the loops that have not been parallelized. Moreover, after seeing how the compiler parallelizes, the programmer can improve upon the parallelization of the compiler. Unfortunately, code produced by previous decompilers is neither portable beyond the parallelizing compiler nor sufficiently natural for understanding parallelism, as shown in Table 1. This is because the primary goals of previous decompilers, including reverse engineering and analysis, do not rely on decompiled code being syntactically correct, recompilable, or natural.

We have identified three core areas that prevent decompilation from enabling collaborative parallelization: lack of explicit parallelism, unnatural control flow translation, and use of artificial variable names. The rest of this section further describes these three roadblocks.

Table 1: Comparison with prior decompiler frameworks. As the first decompiler for collaborative parallelization, SPLENDID emphasizes portability and naturalness for translating parallel code.

Decompiler	Decompilation Level	Primary Goal	Explicit Parallelism Translation				Control Flow Translation		Variable Translation	
			Parallel Runtime Library Call Elimination	Parallel Loop Restoration	Parallel Code Inlining	Pragma Generation	For-Loop Construction	Loop Rotation De-transformation	SSA De-transformation	Source Variable Renaming
Ghidra [1]	binary	Reverse Engineering	✗	✗	✗	✗	✓	✓	n/a	✗
Gussoni et al. [22]	binary	Security	✗	✗	✗	✗	✗	✗	n/a	✗
Chen et al. [12]	binary	Software Maintainance	✗	✗	✗	✗	✗	✗	n/a	✗
SmartDec [17]	binary	Reverse Engineering	✗	✗	✗	✗	✗	✗	n/a	✗
Phoenix [10]	binary	Security	✗	✗	✗	✗	✓	✗	n/a	✗
Hex-rays IDA Pro [53]	binary	Software Validation	✗	✗	✗	✗	✓	✓	n/a	✗
Relyze [33]	binary	Binary Analysis	✗	✗	✗	✗	✗	✗	n/a	✗
Rellic [63, 64]	LLVM-IR	Security	✗	✗	✗	✗	✓	✗	✓	✗
LLVM CBackend [14]	LLVM-IR	Reverse Engineering	✗	✗	✗	✗	✗	✗	✗	✗
SPLENDID (This Work)	LLVM-IR	Collaborative Parallelization	✓	✓	✓	✓	✓	✓	✓	✓

2.1 Lack of Explicit Parallelism

The broad use of OpenMP [44] suggests that it is easier for a programmer to express explicit parallelism through pragmas than controlling threads through calling runtime functions (e.g., pthreads [40]). Prior work lacks the support to encode IR-level parallelism explicitly at the source code level. As a motivating example, Rellic produces code filled with parallelization setup instructions, namely instructions generated to enable parallel execution at lines 3, 7 to 24, and 38 in Figure 1. Some of these parallelization setup instructions are runtime-specific. For example, line 3 of the Rellic-generated code shows the runtime fork call from the LLVM/OpenMP runtime [35], *__kmpc_fork_call*, brought directly from the IR to C. Bringing runtime-specific instructions to the source code restricts portability since the decompiled code can now only be compiled with that specific runtime (e.g., libomp [35] in the motivating example). Moreover, these parallelization setup instructions make produced code unreadable. While the fork call suggests some parallelism, it is not explicit. Without specific knowledge of the OpenMP runtime library designed for LLVM, it can be difficult for a programmer to interpret this line. SPLENDID, however, is designed to produce semantic and portable parallelism.

2.2 Obfuscated Control Flow Translation

Many parallel programming models constrain the structure of the control flow. For example, OpenMP loop-related pragmas only accept loops in canonical *for*-loop format. Thus, failing to produce a canonical Control Flow Graph (CFG) required by the selected parallel programming model for source-level parallelism will result in syntactic errors in the source code. For example, it is syntactically wrong to apply *omp for* to a *do-while* loop.

For parallel programs, loops generated by previous decompilers are often *do-while* loops. This is because loop rotation [34] is a normalization pass that is commonly applied before optimizations (e.g., LLVM *-O1* or higher, and parallelizing compilers such as NOELLE [37] and Polly [21]). Loop rotation transforms each loop into its rotated form in which the exit condition succeeds the loop body. Without further analysis, rotated loops are, at best, decompiled as *do-while* loops with a guard check, as shown in line 25. The guard check did not exist in the original program; it was created by loop rotation to prevent entry to the loop when the initial state of the loop satisfies the exit condition before rotation. This leads to loop-related OpenMP pragmas being unable to be generated because the original *for* loop has been replaced with a loop that OpenMP does not support. SPLENDID instead fully recovers OpenMP-compatible canonical *for* loops.

2.3 Variable Names Irrelevant to Semantics

Prior work produces source code where variable names have no association with the original program semantics [55]. While binaries may still contain debug information that theoretically helps to reconstruct variable names, binary decompilers such as Ghidra were designed for published executables with such data stripped. Even though the IR maps a source code variable to a virtual register with debugging intrinsics, as shown in line 1 of the Parallel LLVM-IR, even fundamental compiler transformations such as Single Static Assignment (SSA) dramatically change the nature of variables in the IR.

First, the number of mappings from a source code variable to virtual registers grows dramatically. This is because promoting a memory reference to a register reference (as done by *mem2reg* in LLVM) may split a single source code variable into multiple instructions connected by a phi instruction to satisfy the SSA form. Moreover, once split, virtual registers may have an overlapping lifetime. That is, one of two virtual registers mapped to the same source code variable may still be alive after the definition of the other (conflict). Two conflicting virtual registers cannot be mapped back to the same source code variable. Additionally, heavily optimized code regions lose such debugging intrinsics because compiler optimizations are performance-driven, lacking the intention to preserve source information which is thought to be unnecessary for improving performance. SPLENDID introduces a new technique to recover the majority of the source code variable names.

3 SPLENDID OVERVIEW

This work introduces SPLENDID, a decompiler framework that produces portable OpenMP code natural for programmer involvement. As shown in Table 2, SPLENDID includes all features necessary for producing portable code. Loop-related transformations, such as Loop Rotation De-transformation, restore loops in IR to canonical *for* loops in source code. Features related to transforming runtime library calls, such as Parallel Runtime Elimination, remove runtime-specific constructs and explicitly express parallelism as OpenMP

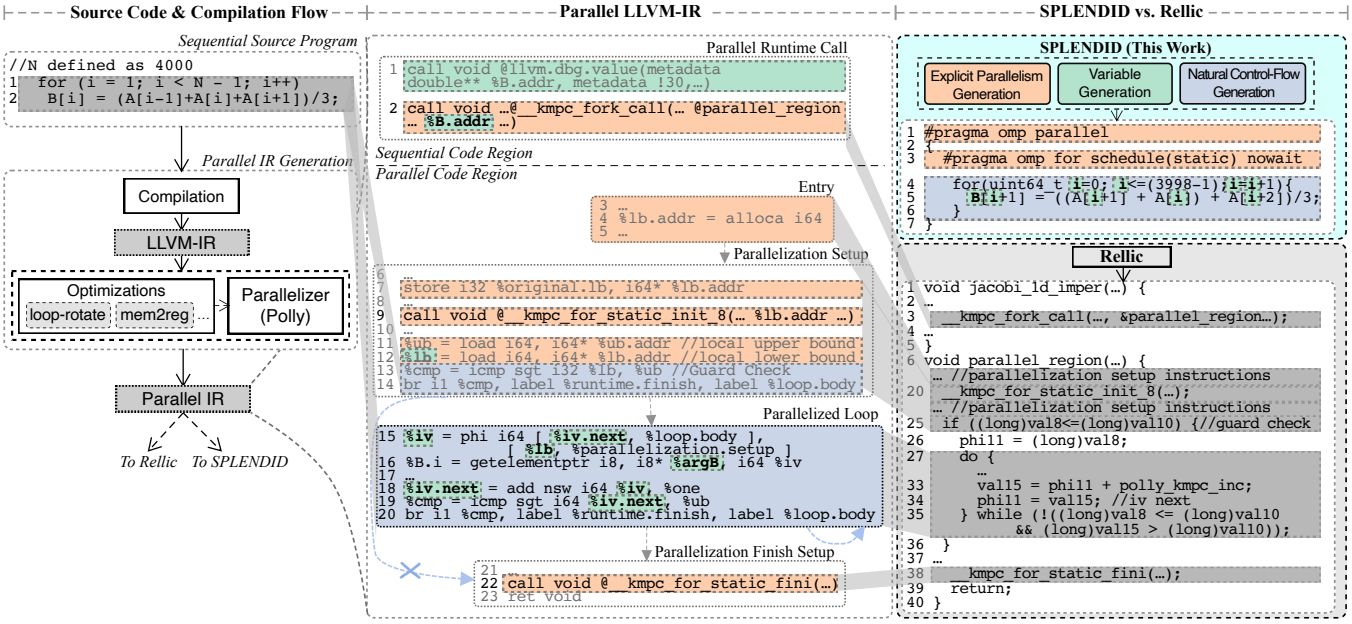

Figure 1: A comparison of code decompiled using Rellic [63, 64] and SPLENDID. The motivating example is a simplified hot loop from `jacobi-1d-imper` in PolyBench [46]. The original sequential code is compiled into LLVM-IR, optimized by LLVM `-O2`, and parallelized using Polly [21]. The resulting parallel LLVM-IR then serves as the input for decompilation. Compared to code produced by Rellic, code produced by SPLENDID is portable to any host compiler and natural.

Table 2: Techniques of SPLENDID to produce portable code that is also natural for manual improvement.

Techniques	Portability	Naturalness
Parallel Runtime Elimination	●	●
Loop Parameter Restoration	●	
Loop Rotation De-transformation	●	●
For Loop Construction	●	●
Parallel Code Inlining		●
Pragma Generation	●	●
SSA Detransformation	●	●
Source Variable Renaming		●

pragmas at the source level. Together, these techniques enable SPLENDID to produce code not only syntactically correct but also compilable universally with any runtime library. In addition to being portable, the same set of techniques also makes SPLENDID-produced OpenMP code more natural, because they restrict code structures (e.g., making loops more natural), and represent parallelism using OpenMP (e.g., eliminating obfuscated runtime function calls). Moreover, SPLENDID chooses variable names that reflect code semantics. The more natural the decompiled code is, the less effort is required from a programmer to understand what and how a compiler parallelizes, and the better chance of additional profitable collaborative parallelization.

By using SPLENDID, a programmer *i)* is freed from parallelizing that which the compiler is capable of parallelizing, *ii)* can make incremental improvements to what the compiler can parallelize, and *iii)* can focus on parallelizing what the compiler cannot parallelize. The rest of this section starts by describing techniques developed to overcome each hindrance identified in Section §2. Then, we present an example demonstrating how each technique in SPLENDID contributes to producing portable and natural source code. Lastly, we present case studies to show how *i)* SPLENDID successfully supports collaborative parallelization, a promising direction in bringing performance in the post-Moore's Law era, and *ii)* despite the enabling power of collaborative parallelization, SPLENDID advances the state of decompilation by making critical optimizations clearer to programmers.

3.1 Explicit Parallel Translation using OpenMP

As far as we know, SPLENDID is the first decompiler that translates parallel IR into portable OpenMP code [44]. This is achieved by first finding loops parallelized from a parallel code region with extraneous parallel execution setup code. Then, the parallel execution setup is removed, namely, instructions that enable a code region to run in parallel, including parallel runtime function calls. With no implementation-specific code at the source code, the decompiled code is portable to any compiler and more natural. After that, SPLENDID generates OpenMP pragmas to replace IR-level parallelism since OpenMP is widely accepted and can be easily understood by developers. Lastly, since the parameters of a parallel loop (e.g., loop bounds and step sizes) are thread-local, loop parameters are restored to the original sequential loop parameters.

3.2 Enhanced Natural Control Flow Translation

Unlike many loop optimizations that bring considerable performance gain, loop rotation, though critical for loop normalization, does by itself not improve program performance but simplifies the implementation of other loop optimizations. Thus, SPLENDID is designed to de-transform rotated counted loops to *for* loops. SPLENDID-generated loops are much more comprehensible since the loop structure is closer to the original source code. Moreover, the well-structured loops generated by SPLENDID make pragma selection easier. As depicted in the motivating example, when the loop is restored to a DOALL *for* loop (with no dependence across iterations), simply applying *#pragma omp for* parallelizes the loop. However, SPLENDID intentionally does not de-transform optimizations other than loop rotation as they do not hinder portability and are critical to performance.

3.3 Natural Variable Reconstruction

Since SSA separates what was originally one source variable into multiple virtual registers (instructions), SPLENDID collapses instructions connected through a phi instruction into one variable. Further, SPLENDID beautifies variables by assigning names to them that are indicative of code semantics. This work proposes relating each instruction with a source variable through mappings extracted from debug information. Unfortunately, because of conflicts, mappings of virtual registers to source variables cannot be directly used to assign variable names. SPLENDID provides a novel verification module that detects and removes conflicts. Details about conflict elimination are described in Section §4.3.2.

If debug information is missing for an instruction, SPLENDID takes one step further and attempts to associate this instruction with debug information from another code region. Specifically, while debug information is preserved throughout transformations to the compiler backend in LLVM, optimizations such as automatic parallelization [3, 21, 37] developed on top of LLVM are not designed with decompilation in mind. Thus, when these optimizations insert new instructions, they may not have precise debug information. To overcome this constraint, SPLENDID assigns source variables to a code region without source information by relating it to a region where source information is present through inlining.

3.4 SPLENDID in Action

Figure 1 shows how each aforementioned technique contributes to the final translation of the example loop. First, all the parallel runtime setup instructions, which are at lines 2–12 and 22 in the parallel LLVM-IR, are used to *i)* restore the parallelized loops to a sequential loop and *ii)* generate OpenMP pragmas for the sequential loop. The parallel code region and its input are some of the inputs to one of the runtime calls, *__kmpc_fork_call* at line 2, through which multiple workers are spawned to execute the parallel region specified as inputs. Parallel regions are functions containing the outlined and parallelized loops from the original code. By recognizing inputs to a fork call, the loops that are parallelized and the original sequential loop parameters are recognized by SPLENDID. The parallelized loop from lines 15 to 20 is then found between the runtime initialization call at line 9 and the finish call at line 22. Since each instance of the parallel region only executes a portion of the parallelized loop, loop parameters such as loop bounds are unique to each worker and are calculated using the original loop parameters as inputs to the initialization call. To restore each parallelized loop to the combined iteration space of all threads, SPLENDID replaces the loop parameters with the values before the initialization call. For example, the lower bound at line 12 is replaced with its original value of 0 at line 7. Once the parallel region is transformed into a sequential loop with OpenMP pragmas, SPLENDID removes all parallelization setup instructions as depicted in gray and inlines the outlined parallel region into the sequential code region.

At this point, the inlined loop is still rotated, and SPLENDID, after verifying the loop is counted by finding the induction variable at line 15, transforms the sequential rotated loop into the *for* loop shown at lines 4 to 6 in the final produced code in Figure 1. Note that SPLENDID removes the guard check at lines 13 and 14 in the parallel LLVM-IR by proving that it is equivalent to the initial exit condition of the transformed *for* loop. Constructing the *for* loop not only improves naturalness but also validates the use of the *#pragma omp for* at line 3, as the pragma can only be used on a *for* loop.

Lastly, SPLENDID collapses and renames virtual registers. SPLENDID removes each phi instruction by replacing its inputs with itself or an expression containing itself. In the motivating example, *iv.next* is replaced with *iv*, the name of the phi instruction, which is then detected as an induction variable and generated as the variable *i* at lines 4 and 5 in the produced code. As the final step, if there is debug metadata relating a source variable name to a virtual register, an instruction, SPLENDID assigns each instruction the source variable name it is related to unless there is a lifetime overlap (such a conflict is described in Section §4.3.2). When an instruction has no debug metadata, such as *argB* from the parallel region at line 16, it can indirectly relate to a source variable through inlining in the following way. First, the sequential code region contains debug metadata that maps instruction *B.addr* to variable B at line 1. Meanwhile, since the parallelized loop is inlined, *argB* is replaced by the input to the parallel region at line 2, which is *B.addr*. Thus, as *B.addr* is mapped to the source variable *B*, so does *argB* after inlining. As SPLENDID finds no conflict with this mapping, variable *B* can safely replace *argB* as the generated variable name at line 5 in the produced code.

3.5 Case Studies

This section presents case studies to demonstrate two use cases of SPLENDID. First, SPLENDID enables compiler-programmer collaborative parallelization, as shown in Table 3 and Figure 2. Second, SPLENDID advances the state of decompilation by presenting natural code in the presence of aggressive compiler transformations, as shown in Figure 3.

3.5.1 Compiler-Programmer Collaboration. Instead of letting the programmer try to optimize the whole program, we propose an alternative approach. That is, before any manual optimization, let the compiler present its parallelization plan to the programmer through SPLENDID.

As shown in Table 3, there is a large overlap between what the compiler and the programmer alone can parallelize. By first letting SPLENDID produce parallel code that is portable to any compiler, programmers are freed from parallelizing loops that are

Table 3: SPLENDID enables the parallelization of all loops that the compiler and programmer can parallelize alone, with reduced manual effort.

Benchmarks	Programmer Parallelized [20]	Compiler Parallelized [21]	Total Parallelizable	Eliminated Manual Parallelization
2mm	2	2	2	2
3mm	3	6	6	3
adi	6	2	7	1
atax	2	1	3	0
bicg	2	1	3	0
doitgen	1	1	1	1
fdtd-2d	4	2	4	2
floyd-warshall	1	1	2	0
gemm	1	4	4	1
gemver	4	4	4	4
gesummv	1	1	2	0
jacobi-1d-imper	2	2	2	2
jacobi-2d-imper	2	2	2	2
mvt	2	3	3	2
syr2k	2	4	4	2
syrk	2	4	4	2
Total	37	40	53	24

parallelizable by a compiler. By knowing what the compiler can parallelize, a programmer can focus on parallelizing loops that the compiler could not parallelize. This way, SPLENDID enables the parallelization of all loops proposed by the programmer and the compiler.

Moreover, whatever is already parallelizable by the compiler may also benefit from the knowledge of a programmer. As shown in Figure 2, the example loop can be conditionally parallelized by the compiler (e.g., Polly [21]) when A and B do not alias. Thus, an aliasing check is injected to provide a fallback to the sequential version of the code when A and B alias. The compiler can emit such aliasing check because *i)* alias analysis is limited (e.g., because it is limited to intra-procedural analysis) to proving that A and B do not alias even if the programmer has only called *MayAlias* with separately allocated units, or *ii)* alias may indeed occur at runtime as the programmer may pass the same pointer to both arguments A and B, as in the case of line 12 in the original code. In the first scenario, a programmer, knowing A and B cannot alias, can remove the sequential version of the code, eliminating the computational overhead from the aliasing check. If the programmer is a compiler writer, he/she no longer needs to search in IR to be informed of the limitation in its compiler analysis. The alias analysis can be improved by looking at decompiled source code. In the second scenario, the programmer, after knowing that the compiler can parallelize the cases when A and B do not alias, can improve the original code by simply restricting the accessing of example code to only when A and B do not alias (i.e., line 1 in (c)) and focus on optimizing cases when A and B must alias in a separate function (i.e., line 6 in (c)). In both scenarios, the programmer is freed from manually parallelizing the example loop under the condition that A and B do not alias. More importantly, such interaction greatly improves the final produced code in both naturalness and performance. Such interaction is only enabled through SPLENDID, as any unnaturalness introduced at the assembly level or by other decompilers can make finding the aliasing check extremely difficult.

(a) Original Code

```
1  void MayAlias(double* A, double* B, double *C) {
2  //... initializations
3   for (i = 0; i < N-1; i++){
4    A[i+1] = M_PI*B[i] + exp(C[i]);
5   }
6  }
7  void main(...){
8   double *A = (double*) malloc(n * sizeof(double));
9   double *B = (double*) malloc(n * sizeof(double));
10  double *C = (double*) malloc(n * sizeof(double));
11  MayAlias(A, B, C);
12  MayAlias(A, A, C);
13 }
```

(b) SPLENDID Output

```
1  void MayAlias(double* A, double* B, double *C) {
2    if ((A+1000) <= B | (B+999) <= (A+1) & (A+1000) <= C | (C+999) <
         = (A+1)) {//Aliasing check
3    #pragma omp parallel
4    {
5     #pragma omp for schedule(static) nowait
6     for(uint64_t i = 0; i<=998; i = i + 1){
7      A[i+1] = (exp(C[i]) + B[i] * 3.1415926535897931);
8     }
9    }
10  } else {
11   for(uint64_t i = 1; i < 999;   i = i + 1){
12    A[(i+1)] = (exp(C[i]) + B[i] * 3.1415926535897931);
13   }
14  }
15 }
```

(c) Specialized Optimization by Programmer

```
1  void NoAlias(double* restrict A, double* restrict B, double* C) {
2   for (i = 0; i < N-1; i++){
3    A[i+1] = M_PI*B[i] + exp(C[i]);
4   }
5  }
6  void <func>_InPlace(double* A, double* C);
7
8  NoAlias(A, B, C);
9  <func>_InPlace(A, C);
```

Figure 2: An example of the programmer removing compiler generated aliasing checks.

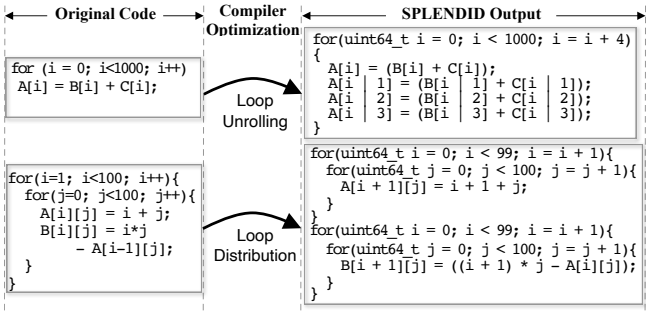

Figure 3: Decompiling loop optimizations using SPLENDID.

3.5.2 Advancement in Decompilation. Natural decompilation goes beyond producing code identical to the original source code. An advanced decompiler should present performance-enabling optimizations the compiler applies in a human-readable way. SPLENDID achieves this by making a trade-off of what to de-transform. Figure 1 has already shown the natural representation of the parallelization of SPLENDID. Instead of aggressively applying de-transformations to all compiler optimizations, SPLENDID chooses to de-transform only peep-hole optimizations that intrude unnaturalness while having little or no influence on performance (e.g., SSA and loop

rotation). This design choice results in aggressive optimizations, such as loop transformations, remaining untouched and presented to the programmer, as shown in Figure 3. Performance engineers can then read the output of SPLENDID and quickly find relevant code properties, such as unrolling factors.

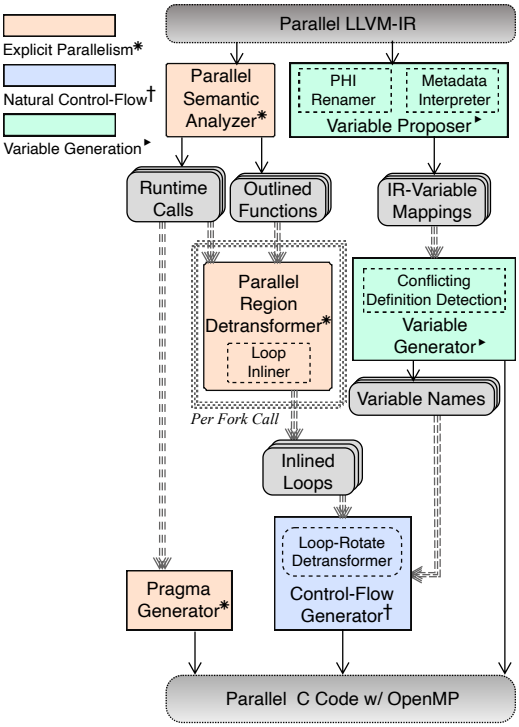

Figure 4: The design and workflow of SPLENDID.

4 DESIGN AND IMPLEMENTATION

This section describes how SPLENDID obtains the features described in Section §3.

4.1 Parallel Source Code Generation

SPLENDID explicitly represents parallel code regions using OpenMP, which consists of the OpenMP pragmas and sequential code regions to which the pragmas are applied. OpenMP requires loops to be strictly structured in counted *for*-loop fashion with no loop-carried dependences. However, beyond the fact that LLVM loops vastly differ from OpenMP-compatible loops, parallel IR contains a large body of low-level OpenMP runtime setup code, making it extremely difficult to extract the parallelized loop. Nevertheless, SPLENDID transforms highly obfuscated IR into portable and natural parallel C code. It starts with the OpenMP Semantic Analyzer.

4.1.1 Parallel Semantic Analyzer. Parallel Semantic Analyzer first collects the runtime calls and then extracts the parallel code regions from the runtime fork calls: *__kmpc_fork_call*. The fork function gets an outlined function as an argument, and when it is called, it creates multiple workers to work on the outlined function simultaneously. The Parallel Analyzer then finds the parallel code region through the outlined function.

4.1.2 Parallel Region Detransformer. Parallelized Region Detransformer *i)* extracts each parallelized loop from the parallel code region, *ii)* restores loop parameters, and *iii)* removes all the parallelization setup instructions such that the output only contains the original loop. To detect a parallelized loop, the Parallelized Region Detransformer searches for loops between a pair of runtime function calls that initializes and ends a parallelized region, such as *__kmpc_for_static_init_8* and *__kmpc_for_static_fini* in Figure 1. Then, loop parameters are restored by replacing them with those used as arguments for the initialization call. Since extraneous instructions can cause an error when placed between *#pragma omp for* and the parallelized loop, the parallelization-related instructions are removed.

The loop is then inlined into the sequential code region. Since the runtime fork call indirectly calls the outlined function, inlining requires that the Loop Inliner replaces arguments passed into the runtime fork call with their corresponding arguments of the outlined function. Lastly, the Loop Inliner replaces the runtime fork call in the sequential region with the transformed sequential loop, eliminating the last runtime-dependent instruction.

4.1.3 Pragma Generator. Explicit parallelism is presented using sequential loops with OpenMP directives. Pragmas are generated from runtime function calls through one-to-one or many-to-one mappings or from static analysis (e.g., private clause). Scheduling policy and chunk size are determined by extracting and interpreting parameters of runtime initialization calls. When more than one correct translation exists, the Pragma Generator uses the most performing pragmas. For example, a pair of *__kmpc_static_for_init_8* and *__kmpc_for_static_fini* with no barrier calls can be transformed into both *#pragma omp for schedule(static)* and *#pragma omp for schedule(static) nowait*. The Pragma Extractor, in this case, produces the latter since there is no implicit barrier.

While most pragmas are generated through runtime calls, the Pragma Generator minimizes the use of clauses to reduce the knowledge required for programmers to interpret code produced by SPLENDID. For example, if the earliest definition of a variable is inside the parallel region, declaring it inside the parallel region by default makes the variable private, thus eliminating the need of using the *private* clause.

4.2 Natural Control-Flow Generation

SPLENDID generates *for* loops which OpenMP requires through de-transforming loop rotation. The Loop Rotate Detransformer first attempts to produce a *for* loop from a well-structured rotated loop by searching for loop parameters, including the induction variable and loop upper and lower bounds. Traditionally, inverting a rotated loop is done by loop peeling, which separates the first iteration from the rest of the loop so that the exit condition can be moved before the loop body. A loop initially rotated from a *for* loop is counted and thus can omit loop peeling by directly changing the upper bound to allow execution of one more iteration since the exit condition is checked before executing the loop body. Inverting the rotated loop creates a *for* loop, but within a guard check created by loop rotation to ensure that if the exit condition before loop rotation fails, the loop is not executed once by mistake. This guard check can be removed if it is verified to be equivalent to the initial

exit condition of the transformed *for*-loop. For example, in Figure 1, the guard check prevents entering the loop if the lower bound is greater than the upper bound in line 13 in the IR. The rotated loop exits if the induction variable initialized with the lower bound incremented by one is greater than the upper bound in line 19. Since loop rotation examines the exit condition one iteration later than the original loop, transforming this exit condition to be used for a *for* loop will make it equivalent to the guard check. Therefore, the guard check in the motivating example can be safely removed by the Loop-Rotate Detransformer.

4.3 Variable Generation

This section describes how variables are generated by combining phi instructions, detecting and utilizing conflict-free debug information, and inlining parallel code regions.

4.3.1 Variable Proposer. Variable generation starts by proposing to replace instructions with variables that reflect original code semantics. The incoming values of phi instructions are proposed to be combined and named with the phi instruction itself. The Metadata Interpreter leverages LLVM-IR metadata containing source variable debug information. The relationship between an IR and a source variable is contained in debug intrinsics encoded as LLVM metadata. As shown in Figure 5, %1 and %2 are both associated with the variable *var* through a debug intrinsic function containing metadata !30. While debug information can be invalidated as optimizations are applied, LLVM guarantees that debug information are correct throughout all the mid-level and backend passes [49], including *mem2reg*. Thus, the Metadata Interpreter can safely rely on debug information. A Metadata Extraction table is built with the debug information, as shown in Figure 5. Since a phi instruction may also be mapped to a source variable when the incoming values are combined for a phi instruction, they are mapped together to the associated source variable.

4.3.2 Variable Generator. While many instructions can be mapped to the same variable, mappings are invalid if instructions mapped to the same variable have a conflict. A conflict in variable naming occurs when a pair of instructions have overlapping lifetimes. For example, a conflict exists between %1 and %2 in Figure 5 since they map to the same variable *var*, and instruction *F* uses %1 after %2 is defined. Renaming them with the same variable results in incorrect execution, as %1 will wrongfully use the value of %2. Thus, the Conflicting Definition Detection module inside the Variable Generator is designed to remove such conflicting mappings. The module detects the most recent variable definitions at every point in the program, as described in Algorithm 1. This is a forward data flow analysis in which a most recent variable to an instruction mapping is generated at this instruction if metadata containing a source variable is available, indicating that the most recent definition of the variable is the current instruction. Simultaneously, the old definition of the variable to which a new definition is generated is killed (i.e., their lifetime ends), as depicted in the Most Recent Variable Definition table in Figure 5.

Once the most up-to-date variable definitions are established at each point of the program, the module then uses it to remove conflicting definitions from the proposed instruction-to-variable map, as described in Algorithm 2. At each use of a proposed variable definition, the algorithm checks if the most up-to-date definition of the proposed variable is indeed the used definition. If not, a conflict is detected, and SPLENDID chooses to remove the most recent mapping to eliminate the conflict arbitrarily. For example, at instruction *F*, the definition at use, %1, is mapped to variable *var* according to the Metadata Extraction Table generated by the Variable Proposer. However, according to the Most Recent Variable Definition table, the most recent definition for *var* is %2 at instruction *F*. Thus, only the %1-to-*var* mapping is valid, and the %2-to-*var* mapping is removed. After Conflicting Definition Detection, the rest of the instruction-to-variable mappings are valid for generating variable names. For example, %3, which also maps to *var*, is not defined before any use of 1 or 2, so it can also be mapped to *var*.

The Variable Generator then generates declarations using the resulting mappings. At the definition or use of a variable, the Variable Generator returns the same value as its declaration by referring to the exact IR-variable mapping shown in Figure 5. As for variables without a mapping, such as %2, they are given the virtual register name as it is unique and somewhat meaningful (e.g., *indvar* tells the programmer this variable is an induction variable).

Algorithm 1: *Most Recent Variable Definitions*

```
Input: IR_Variable_Proposal - Proposed instruction to variable mappings
Result: MR_Var_IR_Maps - Most recent variable definitions at each
        instruction
Var ← getVariable(I, IR_Variable_Proposal);
GEN[I] ← (Var, I);
Kill[I] ← (Var, I_old);
IN[I] ← ∪_{PI∈Pred(I)} OUT[PI];
OUT[I] ← GEN[I]∪(IN[I]-Kill[I]);
MR_Var_IR_Maps ← OUT;
return MR_Var_IR_Maps;
```

Algorithm 2: *Conflicting Definition Removal*

```
Input: IR_Variable_Proposal - proposed instruction to variable mappings
       MR_Var_IR_Maps - Output of Algorithm 1
Result: IR_Variable_Map - validated mappings with conflicting definitions
        removed
for Instruction I in F do
    for Operand op in I.Operands() do
        var ← getVariableName(op);
        if op != MR_Variable_IR_Maps[I][var] then
            IR_Variable_Proposal.erase(pair(op, var));
    end
end
IR_Variable_Map ← IR_Variable_Proposal;
return IR_Variable_Map;
```

5 EVALUATION

This section evaluates the effectiveness of SPLENDID in its claim to produce portable and natural code.

5.1 Experiment Setup

SPLENDID relies on the LLVM Compiler Infrastructure [49] (version 10.0.1) and is built upon the LLVM C Backend [14]. The C backend provides basic support for decompiling C syntax. The program produced by the LLVM C Backend is close to a one-to-one

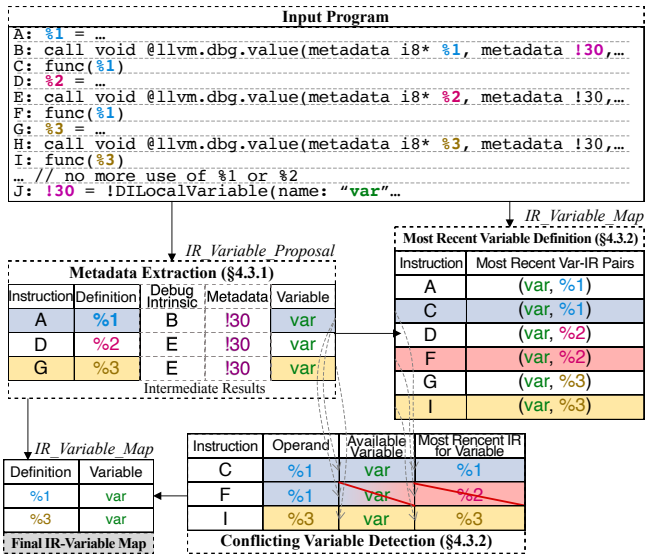

Figure 5: An example of how SPLENDID associates an IR to a source variable (Final IR-Variable Map) through the Metadata Interpreter and Conflicting Definition Detection.

translation from IR instructions to C statements where IR branch instructions translate to C *goto* statements. All the modules described in Section §4 are developed in-house for SPLENDID. The following setups are used for evaluating SPLENDID.

5.1.1 Decompilation Input and Baselines. SPLENDID is evaluated using 16 benchmarks from PolyBench benchmark suite [46]. Other benchmarks in PolyBench are excluded due to the lack of industrial-level robustness in CFG transformations implemented in SPLENDID. To generate parallel IR (i.e., the input to SPLENDID), a benchmark is first compiled to LLVM-IR, optimized with LLVM *-O2*, and parallelized using Polly [21].

Rellic, the state-of-the-art LLVM-to-C decompiler, and Ghidra, a widely used binary-to-C decompiler, are used as baselines. Rellic is the fairest comparison since its input is at the same level as SPLENDID, while the input of Ghidra is binary. Nevertheless, we found Ghidra to be a competitive baseline as it is an industrial standard.

5.1.2 Reference Code. Comparing original sequential code to parallel code generated from parallel LLVM-IR is counter-intuitive when evaluating code naturalness since even manually parallelized code should look different from the sequential code. Thus, we define code naturalness to be how close the decompiled parallel code is to a piece of semantically equivalent hand-written parallel code. To obtain reference code fair for comparison, OpenMP pragmas are manually added into the original sequential source code according to how they are parallelized by Polly to simulate the most natural parallel code that a decompiler can generate using OpenMP without divergence in semantics.

5.1.3 Hardware for Performance Measurement. Performance of all programs are evaluated on a commodity shared-memory machine with two 14-core Intel Xeon CPU E5-2697 v3 processors (28 cores total) running at 2.60GHz (turbo-boost disabled) with 250GB of memory. The operating system is 64-bit Ubuntu 20.04 LTS with GCC 9.4.0.

5.1.4 Metrics. The first metric used is speedup, which shows that SPLENDID-produced code is portable to other host compilers. Then, several naturalness metrics are used to evaluate the claim that since explicit parallelism is expressed through OpenMP directives in SPLENDID, it also brings naturalness on top of portability. First, naturalness is measured using the number of code lines (LoC) since eliminating low-level parallel implementation dramatically reduces LoC. Then, the percentage of variable names restored to the source is provided to show the effectiveness of the variable renaming of SPLENDID (Section §3.3).

Lastly, the BLEU score (BiLingual Evaluation Understudy) [45] is used to measure overall code naturalness. The BLEU score measures the similarity between a reference text to a set of manual translations of the same text. It correlates highly with the human-evaluated quality of natural-language translations [45], which also has been explored recently to evaluate programming languages, specifically in machine learning-based code migration (e.g., source-to-source compilers [2, 16, 28–30]). The use of the BLEU score for formal languages is well established in the literature. CodeXGLUE [36] project developed by Microsoft, for example, uses it to evaluate every source-to-source compiler infrastructure submitted for testing. As in this paper, others [52, 58] use BLEU as the gold standard. However, no other proposed metrics have been found to be practically better. For example, codeBLEU [52] by design is biased towards longer inputs because it can find more matches from the reference. Originally, BLEU score ranges from 0 to 1 with 1 the translated text being identical to a reference. To conform with other literature, this paper uses BLEU-4 score with the score also reported on a scale between 0 and 100. §A illustrates in detail how BLEU-4 score is calculated and its capability in measuring code naturalness. As already pointed out by Tran et al. (in [58]), BLEU does not enforce rigorous word ordering following the syntactic rules of a programming language and thus does not evaluate the correctness of code. Whether code emitted by SPLENDID is syntactically and semantically correct by construction is confirmed by showing a similar speedup to the parallelizing compiler.

5.2 Portability

SPLENDID practically reduces the involvement of programmer in parallelization by replacing original sequential source code with portable parallel source code. Since previous compilers, such as Ghidra and Rellic emit low-level runtime-specific code, the work of a parallelizing compiler, such as Polly, cannot be automatically made available at the source level. Figure 6 shows the result of comparing speedup obtained by Polly-generated binaries and SPLENDID-generated OpenMP code recompiled using Clang and GCC. All binaries were produced with the same optimization level (-O3) and were run 5 times on an near-idle machine. The result shows that SPLENDID-generated code produces identical speedup as Polly, indicating SPLENDID faithfully represents the complete work of Polly. A similar average speedup is obtained when recompiled using GCC and its standard runtime library for OpenMP, libgomp [19], indicating SPLENDID-generated code is compiler-independent. Through

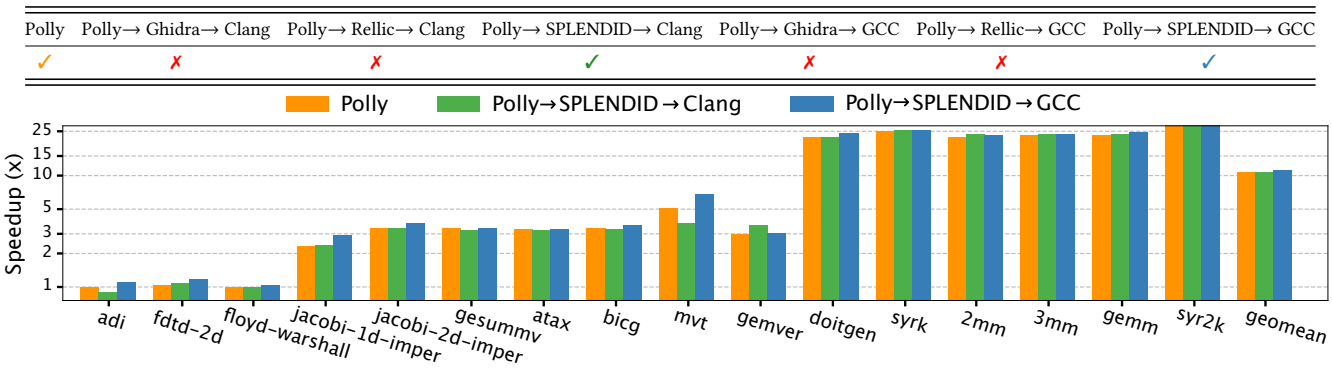

Figure 6: Performance of code decompiled from benchmarks automatically parallelized by Polly using SPLENDID. SPLENDID allows parallelization of Polly to be used by GCC. Polly achieves a geomean speedup of 10.7x on 28 cores. With SPLENDID, GCC also achieves a 11.3x geomean speedup.

SPLENDID, the programmer is freed from parallelizing what a parallelizing compiler like Polly can parallelize, and an average of 11x speedup is made universally available outside of LLVM. The programmer can then choose a compiler with optimizations capable of achieving the best performance. For example, for benchmarks such as *mvt*, GCC produces a noticeable speedup over Clang on the decompiled code.

5.3 Naturalness

The use of OpenMP directives for explicit parallelism brings naturalness to SPLENDID-generated code. On top of that, SPLENDID assigns intuitive variable names to further assist collaborative parallelization.

5.3.1 Naturalness through Explicit Parallelism. Table 4 shows that, by eliminating low-level run-time specific code, parallel representation in SPLENDID-produced code uses less than 13 lines of OpenMP pragmas, including brackets, at least 35x less than naively decompiling parallel execution setup instructions to the source level. The LoC produced by SPLENDID is within 18 LoC difference from the reference code for every benchmark, almost identical to LoC in total with only 0.1x difference, 45x less than the better of the baselines.

The BLEU scores presented in Figure 7 evaluate the overall code naturalness as described in §5.1.4. Using the generated code as the translation under evaluation and the reference code as an instance of natural translation, the BLEU score indicates how close the parallel translation is to manual translation.

We create two variants of SPLENDID to quantify explicit parallelism for naturalness. The first variant, SPLENDID v1, only enables the natural control-flow construction, which contains basic CFG analysis and the novel Loop-Rotation Detransformer proposed in §4.2. On top of natural control-flow construction, SPLENDID v2 enables explicit parallelism translation, representing parallel code regions using inlined sequential loops applied with OpenMP pragmas. Thus, SPLENDID v2 produces code recompilable with any host compiler. All counted loops are generated as *for* loops using SPLENDID v1, yielding an average BLEU score of 1.4, 3.4x higher than the best prior approach in terms of BLEU score, Ghidra. The improvement is not significant because BLEU score focuses on word matching and any improvement in control flow only results in a few keyword differences. SPLENDID v2, however, achieves a much higher BLEU score, indicating that what comes with portability is the massive benefit of naturalness. Code produced by portable SPLENDID scores 21x higher than Ghidra and 43x higher than Rellic due to removing a considerable amount of parallel execution setup code unrelated to original code semantics.

Table 4: Comparison of LoC similarity to reference code. Programs decompiled by SPLENDID contain LoC highly similar to the reference code.

Benchmark	LoC				Parallel Representation (LoC)		
	Ghidra	Rellic	SPLENDID	Ref	Ghidra	Rellic	SPLENDID
2mm	534 (8.1x)	381 (5.8x)	74 (1.1x)	66	343	171	4
3mm	813 (9.0x)	624 (6.9x)	105 (1.2x)	90	620	425	12
adi	311 (5.0x)	371 (6.0x)	70 (1.1x)	62	129	122	4
atax	155 (3.7x)	173 (4.1x)	41 (1.0x)	42	46	49	2
bicg	154 (3.0x)	202 (4.0x)	52 (1.0x)	51	53	52	2
doitgen	442 (9.6x)	307 (6.7x)	58 (1.3x)	46	296	123	2
fdtd-2d	405 (6.8x)	322 (5.4x)	67 (1.1x)	60	132	118	4
floyd-warshall	153 (4.8x)	150 (4.7x)	33 (1.0x)	32	48	49	2
gemm	455 (7.7x)	373 (6.3x)	63 (1.1x)	59	362	262	8
gemver	433 (5.8x)	410 (5.5x)	81 (1.1x)	75	295	275	4
gesummv	130 (2.6x)	155 (3.1x)	41 (0.8x)	50	57	59	2
jacobi-1d-imper	153 (3.8x)	217 (5.4x)	58 (1.4x)	40	70	92	4
jacobi-2d-imper	460 (10.7x)	276 (6.4x)	53 (1.2x)	43	361	142	4
mvt	229 (4.5x)	258 (5.1x)	54 (1.1x)	51	166	185	6
syr2k	458 (7.6x)	379 (6.3x)	62 (1.0x)	60	369	278	8
syrk	400 (7.5x)	357 (6.7x)	59 (1.1x)	53	324	261	8
Total	5685 (6.5x)	4955 (5.6x)	971 (1.1x)	880	3671	2663	76

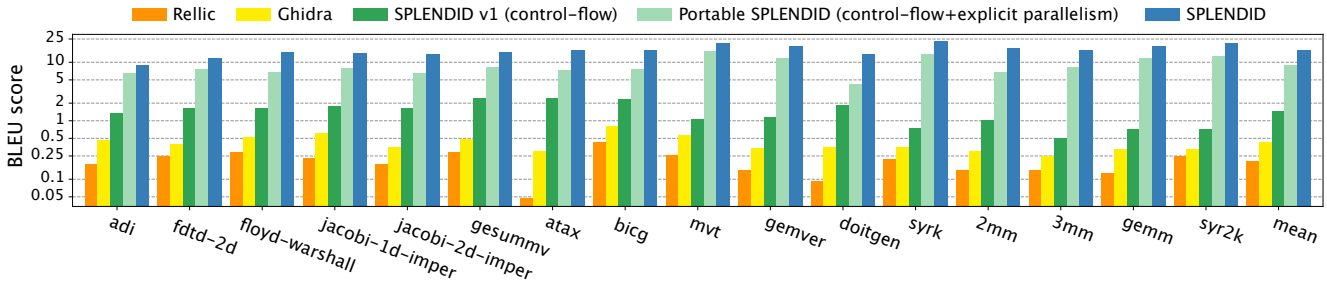

Figure 7: BLEU score comparison of code decompiled using Rellic, Ghidra, and SPLENDID (this work).

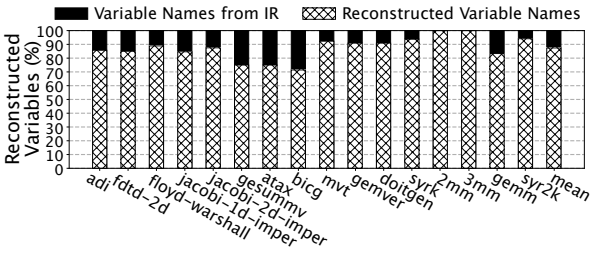

Figure 8: Percentage of variables whose names are reconstructed by SPLENDID.

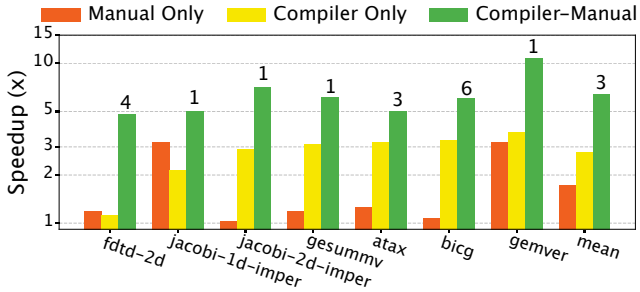

Figure 9: Performance of code with additional manual parallelization after decompiling Polly-parallelized IR using SPLENDID. The numbers represent LoC used to manually parallelize SPLENDID-generated code.

5.3.2 Naturalness through Variable Renaming. SPLENDID-generated code is much more readable because of intuitive variable names. In more detail, Figure 8 shows that, on average, 87.3% of variables are either reconstructed from metadata or inferred through inlining using source variables. Variables that are not reconstructed are because of the loss of source information even before parallelization during the optimization pipeline, such as loop invariant code motion promoting registers and hoisting memory accesses out of the loop. This code hoisting creates an intermediate instruction not associated with any source variable. Since neither Rellic nor Ghidra creates intuitive variable names related to original code semantics, no numbers are provided for prior work. With variable renaming enabled on top of control flow and parallel translation, SPLENDID-generated code achieves an average of 16.4 in BLEU score, 39x times higher than Ghidra and 82x times higher than Rellic.

5.4 Collaborative Parallelization

Portability automatically frees the programmer from parallelizing what Polly can parallelize. As shown in Table 3, among the loops parallelizable by Polly, 60% of what a compiler parallelizes is what the programmer can also parallelize but is freed from doing so. An additional 40% of what Polly parallelizes is beyond the original plan of the programmer and is made available universally for free through SPLENDID.

We found 7 benchmarks among the 16 simple and highly parallelizable PolyBench benchmarks where, surprisingly, neither the programmer nor the compiler achieved the best result, as shown in Figure 9. The manually parallelized PolyBench benchmarks were found on Github by the Cavazos Lab [20]. SPLENDID restores all counted affine *for*-loops, enabling simple manual parallelization on top of SPLENDID-generated parallel code. By simply applying loop distribution (in the case of bicg and atax) and DOALL parallelism to loops that Polly does not parallelize, the speedup is doubled compared to both the compiler and programmer parallelization on its own. This collaborative parallelization is only made practical with SPLENDID.

6 RELATED WORK

Existing tools [13, 26, 48] provide insights into and suggestions from the compiler to the programmer. Intel Advisor, for example, informs the programmer of memory or computation bottlenecks and insights into whether and how to offline code to GPUs. Implicit programming tools, such as COMMSET, provide programmer dependences preventing parallelization. None of these suggestion-based tools have practically reduced the work of a programmer while enabling more parallelism like SPLENDID.

Many advancements in decompilation [22, 39, 53, 54, 63, 64] have greatly improved code naturalness by reducing the usage of *goto* statements. For example, eliminating irreducible graphs [39], diamond-shaped CFGs [22], and many more transformations significantly reduce the number of *goto* statements. However, with loop rotation, loops generated by previous decompilers are often *do-while* loops. For portability, SPLENDID instead de-transforms loop rotation to produce *for*-loops.

Existing LLVM-to-C decompilers [14, 39, 63, 64] produce code that is unnatural. LLVM C Backend [14], the LLVM-to-C decompiler that SPLENDID is built upon, produces assembly-like code with most branches emitted as *goto* statements. Rellic shows no indication of using variable names representative of the code semantics. The C Backend emits source file names and line numbers in its decompiled code using *#line*, a debugging directive. Prior research [15, 32] in debugging has primarily focused on validating debug information instead of using it for variable renaming. SPLENDID, however, directly generates variable names using original source variables.

More work has been devoted to binary-to-C decompilers [12, 17, 22, 39, 54], some of which are integrated into IDEs as part of the debugger (e.g., Ghidra [1], Hex-Rays Decompiler [53], and Relyze [33]). An IDE often has a graphical interface that enables some level of interaction with programmers. Ghidra, for example, allows programmers to rename variables to assist in interpreting code semantics. Likewise, rellic-xref [43], a web interface for Rellic, allows programmers to selectively run some transforms in a user-defined order. However, the kind of interaction is not comparable to the interactive development for parallelization that SPLENDID enables.

Another line of work [16, 29, 30] adopts self-supervised methods in Natural Language Processing using deep learning models. Models are trained using obfuscated source code at a similar level of abstraction to improve code naturalness. Code produced in this approach cannot guarantee correctness and thus requires a manual inspection from programmers. SPLENDID, however, produces code that is semantically correct, portable, and with speedup identical to the underlying automatically parallelized code.

Parallelizing compilers such as QuickStep [38] and Alter [59] insert OpenMP pragmas directly into the original input C code. QuickStep cannot preserve program semantics since it trades accuracy for more parallelism. SPLENDID preserves code semantics in decompilation. Alter requires manual annotations for parallelization and is limited to its own analysis and transformations. SPLENDID, however, is a decompiler that does not target a specific parallelizing compiler. Thus, the performance of SPLENDID-generated code will not be limited by analysis within a single parallelizing compiler.

Source-to-source compilers [5, 27, 41] were designed for code migrations due to naturalness preserved from not lowering to assembly-like IRs. Thus, source-to-source compilers are limited to transformations that do not go beyond the AST level. Some [9, 47, 60] use polyhedral transformations to parallelize code with polyhedral loops. However, parallelization near the source level is not scalable with front-end languages, breaking the ideal source-target independent IR model. Moreover, unlike LLVM IR, rarely is there a community interest in the continued development and maintenance of source-to-source compilers. SPLENDID is easily scalable to other front-end languages as it targets LLVM IR. For the same reason, SPLENDID can be easily supported and maintained within the LLVM community. Furthermore, SPLENDID produces code that is natural and easy for manual code investigation.

7 FUTURE WORK

SPLENDID is the first step in the promising future of practically enabling programmer and compiler collaborative parallelization. As a prototype, SPLENDID only supports OpenMP semantics necessary for Polly, including *parallel*, *for*, *nowait*, *private*, *barrier*, and static schedule, while many other essential features in OpenMP are not supported, such as dynamic scheduling and reduction. Generally speaking, many OpenMP features, such as dynamic scheduling, are lowered into similar constructs involving similar engineering efforts. Clauses such as *reduction* are non-trivial to decompile. However, we see a similar design principle in decompiling parallel regions that can be used to decompile clauses like *reduction*. Moreover, SPLENDID does not preserve comments or handle array flattening when arrays are passed as an argument into a function in LLVM. Future work includes expanding coverage in this way and to handle more sophisticated parallelization, such as speculation [3], PS-DSWP [51], and HELIX [11].

8 CONCLUSION

This work presents SPLENDID, an OpenMP/C decompiler that produces portable and natural code. SPLENDID's naturalness is due, in part, to a novel technique that materializes variable names inferred from the original source code. SPLENDID-produced code achieves a 39x higher average BLEU score than the best prior approach. A decompiler that produces natural parallelized code can enable a more efficient collaborative parallelization effort between the compiler and the programmer. This paper has shown that SPLENDID makes the work of the parallelizing compiler more available to the programmer and frees the programmer from work that can be done automatically. For 7 simple and easily parallelizable programs, in a collaboration enabled by SPLENDID, the compiler and programmer produce code that runs twice as fast as either the compiler or programmer working alone.

9 DATA AVAILABILITY STATEMENT

SPLENDID's artifact evaluation is publicly available and can be downloaded from Zenodo [56].

ACKNOWLEDGEMENTS

We thank members of the Liberty Research Group, the Arcana Lab, and the Argonne National Laboratory for their support and feedback on this work. We also thank the anonymous reviewers for the comments and suggestions that made this work stronger. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract numbers DE-AC02-06CH11357, DE-SC0022138, and DE-SC0022268. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based upon work supported by the National Science Foundation under Grant CCF-2107257, CCF-2118708, CCF-2107042, and CCF-1908488.

A BLEU FOR FORMAL LANGUAGES

Figure 10 illustrates the calculation of the BLEU score. The underlying idea is to build a set of all sub-sequences of length n of a

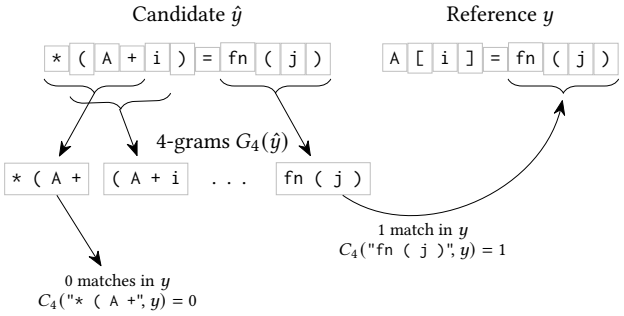

Figure 10: BLUE score calculation

Reference Program

```
for(i=1; i<N-1; i++)
  B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
```

Candidate Programs

(a) Obfuscated Variable Names
BLEU Score: 0.3730

```
for(var0 = 1; var0 < N - 1; var0++)
  var1[var0] = (var2[var0-1] + var2[var0] + var2[var0+1]) / 3;
```

(b) Unnatural Control Flow
BLEU Score: 0.5928

```
if (N - 1 > 0) {
  i = 1;
  do {
    i += 1;
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  } while (i < N - 1);
}
```

(c) No Explicit Parallelism
BLEU Score: 0.3600

```
__kmpc_fork_call(param1, param2, param3, kmp_int32
4, forked_function, param5, A, B, &lb, &ub);

void forked_function(Type1 arg1, Type2 arg2,
double *A, double *B, int *lb, int *ub){
 __kmpc_for_static_init_8(arg1, arg2, 33,
                          lb, ub, 1, 1);
 for (i=*lb; i<*ub; i++)
   B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
 __kmpc_for_static_fini(arg1, arg2);
}
```

Figure 11: A hand-crafted example of BLEU scores reflecting each area of unnaturalness in Section §2.

candidate phrase, called n-grams, and see whether they also occur in a reference phrase. In the case of formal languages, a phrase is a sequence of tokens as detected by the language lexer. The BLEU score is a percentage of matched n-grams relative to the theoretical maximum number of matches (i.e., if the candidate and reference are identical):

$$\frac{\text{Number of matches}}{\text{Theoretical max number of matches}} = \frac{\sum_{s \in G_n(\hat{y})} C(s, y)}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})} \quad (1)$$

A candidate n-gram can occur more times in the reference that in itself; to ensure that the score is in the range $[0, 1]$, the number number of matches that are counted is bounded:

$$\frac{\sum_{s \in G_n(\hat{y})} \min(C(s, \hat{y}), C(s, y))}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})} \quad (2)$$

The final BLEU-4 score is the geometric mean of the n-gram scores of $n = 1, \ldots, 4$. If the candidate phrase is very short, then the denominator will be small and fewer matches be needed to reach a high BLEU score. Therefore, when the candidate phrase is shorter than the reference, an additional brevity penalty is applied[1]. Typically, the final score is presented as a percentage, i.e. multiplied by 100.

The BLEU score for natural languages also allows comparison multiple reference phrases, in which case for each n-gram, the reference phrase with the most matches is used.

This work measures code naturalness using the BLEU-4 score since it is also used in other literature [16, 30] to evaluate formal language naturalness. As shown in Figure 11, unnatural variable names, control flow, and parallelism representation all degrade the BLEU score from 1 (identical to the reference program). While program (a) has a higher 1-gram score with better word-by-word matching, program (b) contains at least an identical loop body to the reference code, resulting in higher 2-gram to 4-gram scores. Thus, programs (b) have shown higher BLEU-4 scores. This means that variable renaming described in Section §3.3] has a significant influence on improving the BLEU score. To show that BLEU scores still reflect improvement in naturalness by constructing natural control flow and explicit parallelism, we thus reported the BLEU score of SPLENDID with variable renaming turned off, namely SPLENDID v1 and Portable SPLENDID, as shown in Figure 7. Overall, the BLEU score of Rellic-produced code in Figure 1 is 0.0035, and SPLENDID-produced code instead scores 0.2932.

REFERENCES

[1] National Security Agency. 2019. Ghidra. https://ghidra-sre.org/.
[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using machine translation for converting python 2 to python 3 code.* Technical Report. PeerJ PrePrints. https://doi.org/10.7287/peerj.preprints.1459v1
[3] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. 2020. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 351–367. https://doi.org/10.1145/3373376.3378458
[4] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I August. 2020. Scaf: A speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 638–654. https://doi.org/10.1145/3385412.3386028

[1] In contrast, the CodeBLEU score is biased towards longer candidates, but does not apply a "verbosity" penalty.

[5] Hamid Arabnejad, João Bispo, João MP Cardoso, and Jorge G Barbosa. 2020. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *The Journal of Supercomputing* 76, 9 (2020), 6753–6785. https://doi.org/10.1007/s11227-019-03109-9

[6] David I. August and Matthew J. Bridges. 2008. The velocity compiler: extracting efficient multicore execution from legacy sequential codes.

[7] Guy E Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. *ACM SIGPLAN Notices* 31, 6 (1996), 213–225. https://doi.org/10.1145/232629.232650

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

[9] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (CC)*. Springer, Berlin, Heidelberg, 132–146. https://doi.org/10.1007/978-3-540-78791-4_9

[10] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[11] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. 2012. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. *IEEE Micro* 32, 4 (July 2012), 8–18. https://doi.org/10.1109/MM.2012.50

[12] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, Zhengwei Qi, Kai Chen, and Haibing Guan. 2010. A Refined Decompiler to Generate C Code with High Readability. In *2010 17th Working Conference on Reverse Engineering*. Institute of Electrical and Electronics Engineers (IEEE), Beverly, MA, USA, 150–154. https://doi.org/10.1109/WCRE.2010.24

[13] Clang. 2023. Expressive diagnostics. https://clang.llvm.org/diagnostics.html.

[14] Julia Computing. 2022. LLVM CBackend. https://github.com/JuliaComputingOSS/llvm-cbe.

[15] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1034–1045. https://doi.org/10.1145/3445814.3446695

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[17] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *2011 18th Working Conference on Reverse Engineering*. IEEE, Limerick, Ireland, 347–356. https://doi.org/10.1109/WCRE.2011.49

[18] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Springer, Berlin, Heidelberg, Budapest, Hungary, 97–104. https://doi.org/10.1007/978-3-540-30218-6_19

[19] GNU. 2022. GNU libgomp. https://gcc.gnu.org/onlinedocs/libgomp/.

[20] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*. IEEE, San Jose, CA, USA, 1–10. https://doi.org/10.1109/InPar.2012.6339595

[21] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1142/S0129626412500107

[22] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *Asia Conference on Computer and Communications Security (ASIA CCS'20)*. Association for Computing Machinery, New York, NY, USA, 637–651. https://doi.org/10.1145/3320269.3384766

[23] Tim Harris and Satnam Singh. 2007. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (ICFP '07)*. Association for Computing Machinery, Freiburg, Germany, 251–264. https://doi.org/10.1145/1291151.1291192

[24] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott Warren. 1994. The D editor: A new interactive parallel programming tool. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, Washington, DC, USA, 733–742. https://doi.org/10.1109/SUPERC.1994.344339

[25] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H Kelm, Isaac Gelado, Sam S Stone, Robert E Kidd, Sara S Baghsorkhi, Aqeel A Mahesri, Stephanie C Tsao, et al. 2007. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*. IEEE, San Diego, CA, USA, 754–759.

[26] Intel. 2022. Intel® advisor User Guide. https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top.html.

[27] Lester Kalms, Tim Hebbeler, and Diana Göhringer. 2018. Automatic OpenCL Code Generation from LLVM-IR Using Polyhedral Optimization. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms* (Manchester, United Kingdom) *(PARMA-DITAM '18)*. Association for Computing Machinery, New York, NY, USA, 45–50. https://doi.org/10.1145/3183767.3183779

[28] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*. Association for Computational Linguistics, Prague, Czech Republic, 177–180. https://aclanthology.org/P07-2045

[29] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. https://doi.org/10.48550/ARXIV.2006.03511

[30] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. *Advances in Neural Information Processing Systems* 34 (2021), 1–18. https://arxiv.org/abs/2102.07492

[31] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. 2012. Parallelizing more Loops with Compiler Guided Refactoring. In *2012 41st International Conference on Parallel Processing*. IEEE, Pittsburgh, PA, USA, 410–419. https://doi.org/10.1109/ICPP.2012.48

[32] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1052–1065. https://doi.org/10.1145/3385412.3386020

[33] Relyze Software Limited. 2022. Relyze. https://www.relyze.com/.

[34] LLVM. 2023. LLVM loop terminology (and canonical forms). https://llvm.org/docs/LoopTerminology.html#rotated-loops

[35] LLVM/OpenMP. 2023. LLVM/OpenMP 15.0.0git documentation. https://openmp.llvm.org/design/Runtimes.html.

[36] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021), 1–14. https://arxiv.org/abs/2102.04664

[37] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, et al. 2022. NOELLE Offers Empowering LLVM Extensions. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, IEEE, Seoul, Korea, 179–192. https://doi.org/10.1109/CGO53902.2022.9741276

[38] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 1–26. http://dx.doi.org/10.1145/2465787.2465790

[39] Simon Moll. 2017. AST - Extractor for LLVM (axtor). https://github.com/cdl-saarland/axtor.

[40] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[41] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. 2011. Source-to-Source Code Translator: OpenMP C to CUDA. In *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, Banff, AB, Canada, 512–519. https://doi.org/10.1109/HPCC.2011.73

[42] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[43] Trail of Bits Inc. 2022. rellic-xref. https://github.com/lifting-bits/rellic/tree/master/tools/xref.

[44] OpenMP Architecture Review Board. 2007. OpenMP Application Program Interface.

[45] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[46] Louis-Noël Pouchet. 2021. Polybench/C. http://web.cs.ucla.edu/~pouchet/software/polybench/.

[47] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. 2010. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *Conference on Supercomputing (SC'10)*. IEEE Computer Society Press, New Orleans, LA, 1–11. https://doi.org/10.1109/SC.2010.14

[48] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. 2011. Commutative Set: A Language Extension for Implicit Parallel Programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/1993498.1993500

[49] LLVM Project. 2022. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[50] Arun Raman, Jae W. Lee, and David I. August. 2012. From Sequential Programming to Flexible Parallel Execution. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Tampere, Finland) *(CASES '12)*. Association for Computing Machinery, New York, NY, USA, 37–40. https://doi.org/10.1145/2380403.2380417

[51] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 114–123. https://doi.org/10.1145/1356058.1356074

[52] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020), 1–8. arXiv:2009.10297 https://arxiv.org/abs/2009.10297

[53] Hex-Rays SA. 2021. Hex-Rays Decompiler - User Manual. https://www.hex-rays.com/products/decompiler/manual/.

[54] Snowman Decompiler 2021. Snowman Decompiler. https://github.com/yegord/snowman.

[55] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167.

[56] Zujun Tan, Yebin Chon, Michael Kruse, Johannes Doerfert, Ziyang Xu, Brian Homerding, Simone Campanoni, and David I. August. 2023. *SPLENDID: Supporting Parallel LLVM-IR Enhanced Natural Decompilation for Interactive Development.* Zenodo. https://doi.org/10.5281/zenodo.7592823

[57] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, Springer, Berlin, Heidelberg, 179–196. https://doi.org/10.1007/3-540-45937-5_14

[58] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU Score Work for Code Migration?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE Press, Montreal, Quebec, Canada, 165–176. https://doi.org/10.1109/icpc.2019.00034

[59] Abhishek Udupa, Kaushik Rajan, and William Thies. 2012. ALTER: exploiting breakable dependences for parallelization. *ACM SIGPLAN Notices* 47 (08 2012), 480. https://doi.org/10.1145/2345156.1993555

[60] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (jan 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[61] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. 2007. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (San Jose, California, USA) *(PPoPP '07)*. Association for Computing Machinery, New York, NY, USA, 79–89. https://doi.org/10.1145/1229428.1229443

[62] Peng Wu, Arun Kejariwal, and Călin Caşcaval. 2008. Compiler-driven dependence profiling to guide program parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, Springer, Berlin, Heidelberg, 232–248. https://doi.org/10.1007/978-3-540-89740-8_16

[63] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 158–177. https://doi.org/10.1109/SP.2016.18

[64] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS Symposium 2015 (NDSS '15)*. Internet Society, San Diego, CA, USA, 1–15. https://doi.org/10.14722/ndss.2015.23185

Received 2022-10-20; accepted 2023-01-19