<u>N</u>OELLE <u>O</u>ffers <u>E</u>mpowering <u>LL</u>VM <u>E</u>xtensions

Angelo Matni[†], Enrico Armenio Deiana[†], Yian Su[†], Lukas Gross[†], Souradip Ghosh[†],
Sotiris Apostolakis[§], Ziyang Xu[§], Zujun Tan[§], Ishita Chaturvedi[§], Brian Homerding[†], Tommy McMichen[†],
David I. August[§], Simone Campanoni[†]
[§]Princeton University, USA
[†]Northwestern University, USA

***Abstract*—Modern and emerging architectures demand increasingly complex compiler analyses and transformations. As the emphasis on compiler infrastructure moves beyond support for peephole optimizations and the extraction of instruction-level parallelism, compilers should support custom tools designed to meet these demands with higher-level analysis-powered abstractions and functionalities of wider program scope. This paper introduces NOELLE, a robust open-source domain-independent compilation layer built upon LLVM providing this support. NOELLE extends abstractions and functionalities provided by LLVM enabling advanced, program-wide code analyses and transformations. This paper shows the power of NOELLE by presenting a diverse set of 11 custom tools built upon it.**

I. Introduction

The compiler community is on the front lines of satisfying the continuous demand for computational performance and energy efficiency. The focus of compiler advancements is shifting beyond peephole optimizations and the extraction of instruction-level parallelism. More aggressive optimizations and more sophisticated, wider scope analyses are required to accommodate the needs of emerging architectures and applications.

Modern compilers use low-level intermediate representations (IR) to perform optimizations that are language-agnostic and architecture-independent, such as LLVM IR from the LLVM compiler framework [1], [2] and GIMPLE from GCC [3]. Low-level IR, along with a set of low-level abstractions built around it, is designed to aid program analyses and optimizations and has shown its value for peephole optimizations and extraction of ILP. However, low-level abstractions are not enough for advanced code analyses and transformations. Consider automatic parallelization, one of the most powerful program optimization techniques, exists only in a basic form [4]–[6], or does not exist at all in most general-purpose compilers. This paper shows that, with proper abstractions, a daunting automatic parallelization transformation can be implemented in fewer than a thousand lines of code.

Advanced code analyses and transformations go hand in hand with higher-level abstractions, as shown by many existing compilers and frameworks. Several compiler infrastructures that support automatic parallelization [7]–[9] all operate on high-level abstractions and perform source-to-source translation. The recent success of domain-specific compilers/frameworks also proves the importance of high-level abstractions for optimizations by uncovering optimization opportunities at a domain-specific graph or operator level [10], [11]. However, these compilers limit themselves to specific program languages or problem domains, and miss opportunities only presented by low-level IRs, including more fine-grained operations and more canonical code patterns.

The combination of higher-level abstractions and lower-level IR is the key to advanced program analyses and optimizations. The claim can be found in the SUIF compiler [12], which provides low-level IR as well as higher-level constructs [13]; and the IMPACT compiler [14], which provides hierarchical IRs to enable optimizations at different levels. Despite the claim, we are not aware of actively-maintained domain-independent compilers that fulfill this combination.

While LLVM has become the de-facto compiler infrastructure to build upon, it does not provide many essential abstractions for advanced analyses and transformations, including abstractions designed to describe properties of a wider code scope (e.g., dependence graph of a whole program) or functionalities for advanced code transformations (e.g., task creation, scheduling a whole loop). These abstractions can ease the implementation of new transformations and make existing code transformations available in LLVM more powerful.

We propose a new open-source compilation layer called NOELLE that delivers abstractions and functionalities for advanced code analyses and transformations. To demonstrate the importance of NOELLE, we have implemented 11 advanced code transformations, nine of which need only a few lines of code. Only one of these transformations is currently available in LLVM (i.e., loop invariant code motion). We will show that our version is significantly more powerful and the implementation is more elegant than the LLVM counterpart. The other 10 transformations are missing in LLVM because they are challenging to implement with the abstractions currently provided by LLVM. Finally, each NOELLE abstraction is used by most of the 11 significantly-different code transformations, which demonstrates how versatile NOELLE abstractions are.

We implemented a variety of code transformations upon NOELLE: a few parallelizing compilers, a Pseudo-Random value generator selector, a comparison optimization for timing speculative micro-architectures, dead function elimination, a memory guard optimization, a code analysis and transfor-

This project was supported by the United States National Science Foundation via grants 2107042, 2118708, 2119069, 2028851, 1908488, 1814654, 1763743, by the United States Department of Energy via the grant DE-SC0022268, and by ARM Ltd.

mation to replace hardware interrupts, and loop invariant code motion. These tools are challenging to implement with only the low-level abstractions provided by LLVM, however NOELLE enables a powerful and elegant implementation. We tested all these tools on 71 benchmarks from four benchmark suites (SPEC CPU2017, PARSEC 3.0, MiBench, and PolyBench). All these tools improve the quality of the code generated by LLVM with its highest level of optimization. Finally, the high heterogeneity between these 11 custom tools suggests NOELLE provides general abstractions and support for a wide variety of advanced code analyses and transformations. Finally, we have released NOELLE publicly [15].

This paper:

- Introduces NOELLE, a robust open-source domain-independent compilation layer built upon LLVM,
- Describes the abstractions and functionalities provided by NOELLE to ease development of advanced code transformations and analyses (Section II-B),
- Presents the tools provided by NOELLE to ease the deployment of custom compilation tool-chains (Section II-C),
- Describes a diverse set of 11 custom tools built upon NOELLE (Section III) to highlight NOELLE's benefits,
- Evaluates the importance and accuracy of NOELLE's abstractions (Section IV), and
- Further motivates the need for NOELLE by comparing it with prior work (Section V).

II. NOELLE

We now describe NOELLE, its abstractions, and its tools.

A. NOELLE in a Nutshell

The goal of NOELLE is to provide abstractions and functionalities that enable and simplify the implementation of complex code analyses and transformations, referred to as custom tools, which target wide program scopes. Custom tools built upon NOELLE include LLVM passes that work at the IR level to perform their code analyses and transformations. Allowing these custom tools to be easily implemented and maintainable requires simple, domain-independent abstractions powered by either accurate low-level code analyses or complex low-level code transformations. NOELLE provides such abstractions (Section II-B) with a modular design allowing its users to pay only the cost of creating the abstractions requested.

NOELLE's abstractions are powered by code analyses, some of which are provided by third parties. For example, the call graph abstraction NOELLE provides is computed by relying on the PDG, which is computed from several alias analyses implemented by external codebases (SCAF [18] and SVF [19]) as well as those provided by LLVM. Moreover, NOELLE's modular design makes it easy to extend the list of external code analyses that power NOELLE's abstractions. NOELLE also provides tools (Section II-C) to simplify the implementation of user-specific compilation flows.

Most abstractions NOELLE provides are either not available in LLVM or they significantly extend those provided by LLVM. The remaining abstractions generalize the LLVM ones (e.g., IV). NOELLE's abstractions and related functionalities are listed and briefly described in Table I. Table II describes the importance of the extra functionalities NOELLE provides compared to LLVM's ones.

Input and Output. The input of a compilation flow built upon NOELLE is the source code of a program and optionally, a set of training inputs that could be used for profile-guided or autotuning-based custom tools. The output is a binary for a target architecture supported by vanilla LLVM backends.

An Example of Compilation Flow. NOELLE enables its users to deploy custom compilation flows by providing a set of tools, described in Section II-C. Next, we describe an example of a compilation flow built using NOELLE's tools (Figure 1). This is the compilation flow used by the custom tool HELIX (further described in Section III).

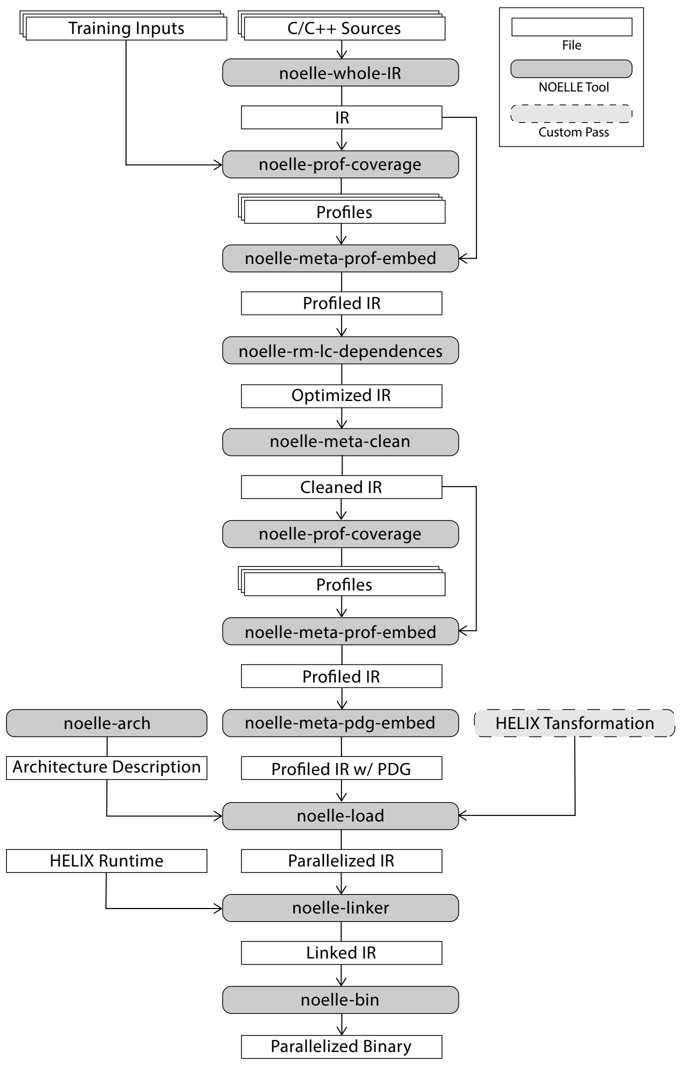

Fig. 1: Compilation flow of the HELIX custom tool using NOELLE tools and a custom pass, HELIX Transformation. Figure 2 shows how to build HELIX Transformation using NOELLE abstractions.

Each source file composing a program being compiled is consumed by `noelle-whole-IR`, which outputs a single LLVM IR file that includes the whole program's code as

TABLE I: FUNCTIONALITIES AND ABSTRACTIONS ADDED BY NOELLE COMPARED TO LLVM

Abstraction	*LLVM functionalities*	*NOELLE additional functionalities*
Data-flow engine (DFE)	Not provided	Optimized engine to evaluate user-provided data flow equations. Optimizations described in well-known compiler books [16], [17] are all implemented.
Environment (ENV)	Not provided	Variables needed by a code region (e.g., a task) to execute (live-ins and live-outs). Functionalities to transform code to propagate live-ins and live-outs between functions.
Task (T)	Not provided	Code region (with its inputs/outputs and explicit possible predecessors/successors) that can be asynchronously executed.
Reduction (RD)	Supports only single-use reducible variables	Identifies and support reducible variables of a loop independently on its uses.
Scheduler (SCD)	Only available in the back-end and only within a single basic block	Several schedulers specialized at different code region granularities (e.g., scheduler of loops, instruction schedulers within and across basic blocks).
Profiler (PRO)	Profilers at the instruction or branch edge granularities.	Adds profilers of loops (e.g., how many iterations a loop execute per invocation), functions (e.g., how often a function is invoked directly or indirectly by another one), and SCCs (e.g., how many instances an SCC has executed). Adds iterators to consider only code regions with specific characteristics (e.g., loops with run-time coverage higher than a threshold).
PDG	To appear in the next stable version. Dependence existence only within a single function. No additional information attached to dependencies. A dependence cannot cross a single function.	Dependence information within a configurable scope, up to the entire program. Provides a hypergraph to describe each dependence at different granularity (e.g., a dependence between two loops expands to the dependences between the single instructions involved).
SCCDAG	To appear in the next stable version. SCCDAG derived from the data dependence graph (DDG) rather than the PDG, so it does not include dependencies of all types. No additional information attached to dependencies.	Adds the capability to compute SCCDAG from any PDG configuration (e.g., DDG, control dependence graph, PDG of a given scope). Adds attributes that describe characteristics of each SCC (e.g., induction variable relationship, side-effect free, self-commutative, loop-carried). Adds iterators to traverse a configurable set of nodes (e.g., to iterate over only SCCs with a loop-carried control dependence).
Call graph (CG)	Partial call graph created using only direct call/invoke instructions. Absence of an edge means call may or may not exist.	Complete call graph considering both direct and indirect call/invoke instructions.
Loop structure (LS)	LLVM's class `Loop` describes the structure of a loop (e.g., its exits). It can only provide information about a single function.	Adds information about the shape of the loop (e.g., while-shape, do-while shape). Supports queries involving multiple functions.
Invariant (INV)	Only instructions outside a loop, arguments, and constants are considered invariants	Instructions within a loop that do not change value among iterations are identified as loop invariants.
Induction variable (IV)	Induction variables only for do-while loops	Induction variables of all loops, while and do-while ones.
Induction variable stepper (IVS)	Not provided	Modifies the code of a loop to implement a change in step value of an induction variable.
Loop Content (LC)	Not provided	A loop with its dependence graph, its SCCDAG, its invariants, its induction variables, and its loop structure information.
Forest (FR)	Forest of loop trees for a single function.	Forest of loop trees for the whole program. It adjusts when a node is deleted to keep the connections between its parent and its children.
Loop transformer (LT)	Some loop transformations (e.g., loop unrolling). They are not organized under a single abstraction.	Adds loop splitting, translating do-while loops into while form, and creating new loops. Extends loop fusion. Added transformations and LLVM ones are organized into a single abstraction.
Islands (ISL)	Not provided	Identifies the disconnected sub-graphs of a graph (e.g., call graph, PDG).
Architecture description (AR)	Alignment requirements and size of builtin variables	Description of the underlying architecture in terms of logical/physical cores, NUMA nodes. It also provides the measured latencies and bandwidths between pairs of cores.

well as options to use to generate the final binary (e.g., the libraries to link with). Then, using training inputs given to NOELLE, `noelle-prof-coverage` runs several profilers to collect statistics about the single IR file's execution. These statistics include the hotness of code regions (e.g., a loop, a basic block), loop-specific information (e.g., the total number of iterations of a loop, the average number of iterations per invocation of a loop), and function-specific information (e.g., number of invocations of a function, the average number of recursive calls of a recursive function). The program's profiles are then embedded into the IR file by `noelle-meta-prof-embed`. The generated IR is consumed by `noelle-rm-lc-dependencies`, which applies a set of code transformations that aim to reduce loop-carried data dependencies in hot loops (i.e., the minimum hotness required to consider a loop). The generated IR is now more amenable to loop-centric code parallelization techniques. The tool `noelle-meta-clean` cleans all NOELLE-specific metadata from the IR file. Then, `noelle-prof-coverage` and the tool `noelle-meta-prof-embed` re-generate and embed the program's profiles, respectively. Then, `noelle-meta-pdg-embed` computes the program dependence graph (PDG) and embeds it as metadata inside the IR file. The `noelle-arch` computes architecture-specific profiles (e.g., communication latency between cores). Its output is used by the HELIX transformation. Finally, the `noelle-load` tool is invoked, which loads in memory NOELLE's compilation layer, to run the HELIX transformation. The HELIX transformation relies on NOELLE's abstractions to parallelize hot loops. The generated parallelized IR file is then consumed by `noelle-linker`, which embeds the HELIX-specific runtime into the IR. Finally, `noelle-bin` generates the parallel binary.

B. NOELLE's Abstractions

Next, we describe the abstractions that NOELLE provides to its users. NOELLE's abstractions (summarized by Tables I, II) are demand-driven and customizable to preserve compilation time and memory. Hence, users only pay for the abstractions

TABLE II: NOELLE's ABSTRACTIONS AND FUNCTIONALITIES ENABLE SEVERAL ANALYSES AND TRANSFORMATIONS.

Abstraction	*Importance of the additional NOELLE's functionalities*
Data-flow engine (DFE)	Enables data-flow analyses to be implemented with a simple declaration of their equations.
Environment (ENV)	Simplifies the extraction of a code region from its context where it is defined. For example, ENV simplifies a loop to be outlined into another function. Also, when ENV is coupled with Task, it enables parallel executions between code regions and/or scheduling of entire code regions.
Task (T)	Enables the design of task-centric asynchronous execution models.
Reduction (RD)	Enables transformations to avoid performance degradation due to dependencies involving reducible variables.
Scheduler (SCD)	Enables transformations to go beyond peephole optimizations like scheduling a whole loop within a function.
Profiler (PRO)	Enables transformations to understand the dynamic aspects of code regions going beyond single instructions.
PDG	Knowing the dependence graph up to the whole program enables program-wide code analyses and transformations. Providing the ability to compute the dependence graph at different granularities enables users to pay only what they need. Dependencies in NOELLE's PDG come with a rich set of attributes describing its characteristics; for example, loop-carried with dependence distance, memory allocation location for memory accesses, commutativity of a dependence. This enables transformations to specialize depending on dependence's characteristics (e.g., a code scheduler capable to consider both orders of a pair of instructions connected by a commutative dependence). Finally, NOELLE's PDG provides several iterators for traversing a configurable set of dependencies; these iterations simplify user's code.
SCCDAG	Understanding the specific characteristics of an SCC enables transformations to specialize their handling of dependence cycles (e.g., enabling out-of-order executions between dynamic invocations of a commutative SCC).
Call graph (CG)	A complete call graph enables transformations to act upon the lack of edges between two functions (e.g., if a function has no incoming edges, then it can be deleted as unreachable).
Loop structure (LS)	Simplifies the implementation of inter-procedural transformations that need knowledge of loops of different functions.
Invariant (INV)	Increases significantly the number of loop invariants making related transformations more powerful.
Induction variable (IV)	Enables transformations that must work with while-shape loops (e.g., parallelization).
Induction variable stepper (IVS)	Simplifies transformations that change how to traverse the loop's iteration space.
Loop Content (LC)	Allows users to have all information about a loop to be accessible through a single abstraction.
Forest (FR)	Enables transformations to target loops in different functions at the same time.
Loop transformer (LT)	Simplifies transformations that need to modify a loop using a sequence of complex loop transformations (e.g., splitting a loop into two and then take the second one and merge it with another loop).
Islands (ISL)	Simplifies transformations that need to know whether two nodes are connected directly or indirectly (e.g., dead function elimination).
Architecture (AR)	Enables transformations to specialize the generated code depending on platform-specific characteristics of its cores and NUMA zones.

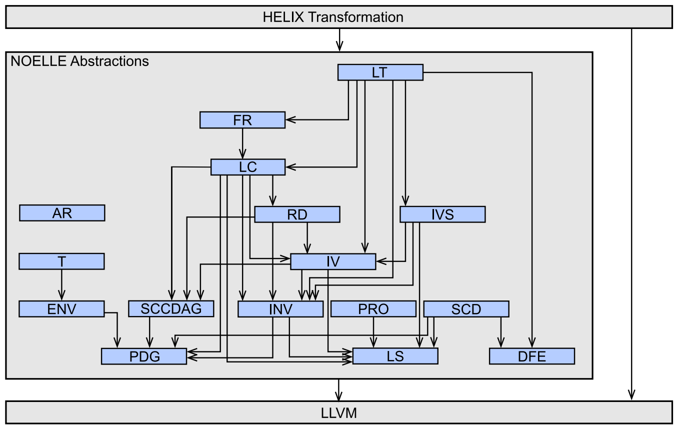

Fig. 2: Arrows in the graph describe the dependence between analyses used by the HELIX transformation.

they need. In other words, if a user does not need the program dependence graph (PDG), then it will not pay the cost of analyzing the program to compute its dependencies. Also, if an user needs only the PDG of a loop, then it will only pay the cost of running loop-specific dependence analyses for that loop. Table I describes the additional functionalities NOELLE provides compared to LLVM 12. Table II describes what these additional functionalities enable.

PDG. NOELLE provides the Program Dependence Graph (PDG) representation of a program [20]. We made the design choice of enabling users to obtain a PDG at different code granularities (e.g., dependencies between single instructions or single loops). This enabled NOELLE's users to specialize their use of NOELLE for their specific needs. This is obtained by providing the ability to change the semantics of a node in the graph (e.g., single instruction, single loop, single SCC) as well as by allowing users to request the PDG of a subset of the program's code.

The PDG builds upon NOELLE's *dependence graph*, a templated class designed to represent a generic graph of directed dependencies between nodes. What constitutes a node is decided when the class is instantiated. For example, an instruction-level PDG instantiates this templated class with the LLVM instruction class. Hence, the nodes of this PDG are the instructions of a program. Independently on the node choice, each edge of a PDG contains a rich set of attributes to differentiate between dependencies (e.g., control and data). Data dependencies are further characterized by the dependence type (Read-After-Write, Write-After-Write, Write-After-Read), whether it is loop-carried, dependence distance, whether it is commutative, whether it is a memory or register dependence, and whether it is an apparent (may) or actual (must) dependence [21]. Finally, each edge can contain sub-edges to describe that dependence at a lower granularity. For example, an edge in an SCC-level PDG contains sub-edges, which represent the instruction-level dependencies that are responsible for this SCC-level dependence.

An analysis or transformation built upon NOELLE can use the PDG abstraction to create function dependence graphs and loop dependence graphs. The former refers to dependencies only between the instructions of a function and it is computed directly from the PDG. The latter is a dependence graph of a specific loop and it uses additional loop-centric analyses. In more detail, when a pass requests the loop dependence graph of a loop from a PDG, NOELLE runs loop-centric analyses to refine, and improve the precision about, the dependencies that are included in the PDG for the specific loop in-question. NOELLE computes the PDG without using loop-centric memory analyses and only use those when requested because loop-centric memory analyses are the slowest ones and most users do not need the most accurate set of dependences for cold loops. Finally, NOELLE enables users to compute a PDG of an arbitrary set of instructions.

Users of the PDG abstraction often want to not only know about the nodes of a dependence graph that belong to a related code region (e.g., instructions of a loop for a loop dependence graph) but also about the inputs and the outputs of the graph. For example, a parallelizing code transformation of a loop needs to know the live-in and live-out sets of the target loop. Because of this need, the *dependence graph* offers two sets of nodes, both internal and external. The former belong to the related code region; the latter represents the live-ins and live-outs of that code region. The computation of both sets of nodes is computed by NOELLE when a pass requests either a loop dependence graph or a function dependence graph.

SCCDAG. Advanced code transformations like parallelization techniques can be implemented as different strategies to schedule instances of the nodes that compose the SCCDAG of a loop [22], [23]. For instance, HELIX distributes instances of a given SCCDAG node around the cores. DSWP instead distributes nodes of an SCCDAG between cores while keeping all instances of a given node within the same core. Hence, an important abstraction is the SCCDAG, which NOELLE provides. Building this abstraction upon NOELLE's *dependence graph* enabled the SCCDAG of a given loop to be a complete description of loop dependencies, including those with the rest of the program, and users can now change the granularity of the graph from SCC to instructions if needed.

NOELLE includes code analyses (e.g., commutative analysis, induction variable analysis) that are used when an SCCDAG is computed. Such analyses provide a rich description of each SCC. In more detail, a node of an SCCDAG can be `Independent`, `Sequential`, or `Reducible`. This categorization of a node n depends on the relation between the instructions' dynamic instances included in n for a given loop invocation. If all these instances are independent of each other, then n is tagged as `Independent`. If an instance of an instruction of n depends on another instance of an instruction of n, then this node is tagged as `Sequential`. Furthermore, if there are dependencies between instances of n, but they are reducible by a reduction code transformation (e.g., by cloning the defined variable `s` in `s += work(d)`), then n is tagged as `Reducible`, and the related reduction is described within the node. Finally, each SCC can be ordered or commutative depending on whether the dynamic instances of the related SCC must be preserved or not.

TABLE III: DEPENDENCIES BETWEEN NOELLE'S ABSTRACTIONS

Abstraction	*Depends on*
DFE	
ENV	PDG
T	ENV
RD	SCCDAG, INV, IV
SCD	PDG, LS, DFE
PRO	LS
PDG	
SCCDAG	PDG
CG	PDG
LS	
INV	PDG, LS
IV	LS, INV, SCCDAG
IVS	LS, INV, IV
LC	LS, PDG, IV, INV, SCCDAG, RD
FR	LC, CG
LT	FR, LC, DFE, IV, IVS, INV
ISL	PDG, CG
AR	

Call graph (CG). NOELLE provides the complete *call graph* of a program where nodes are functions, and edges indicate a given function invokes another. This graph is complete so the lack of an edge means a function cannot invoke another. This abstraction relies on the PDG to compute the possible callees of an indirect call. Edges of NOELLE's call graph can be must or may depending on whether a given caller-callee relation is proved to hold or not. Each edge has sub-edges to indicates with which specific instructions a caller invokes another function. Finally, CG can compute the set of disconnected islands of such a graph.

Environment (ENV). NOELLE offers the *Environment* abstraction, which is an array of pointers of variables. Variables within an Environment represent the incoming and outgoing values for a set of instructions. Finally, NOELLE provides *Environment Builder* to create, modify, and query environments.

Task (T). NOELLE offers the *Task* abstraction to describe a code region that runs sequentially. Parallelization techniques use the above abstraction in the following way. Nodes within an SCCDAG are partitioned into tasks. An Environment is created for each task. At runtime, tasks are submitted to a thread-pool, which will run them in parallel across the cores. The explicit forwarding of data values between tasks is performed with the knowledge provided by the Environment.

Data flow engine (DFE). NOELLE provides a *data flow*

TABLE IV: NOELLE'S TOOLS

Tool	*Description*	*Depends on*
`noelle-whole-IR`	Generate a single IR file from C/C++ source files embedding the compilation options as metadata inside the generated IR file.	
`noelle-norm`	Normalize an IR file (e.g., all loops will be in LCSSA form). NOELLE's abstractions are computed assuming the IR analyzed is normalized using this tool.	
`noelle-rm-lc-dependences`	Transform loops to remove as many loop-carried data dependences as possible. This tool generates normalized IR by invoking `noelle-norm`.	SCCDAG, CG, L, PRO, FR, LB
`noelle-prof-coverage`	Inject code into the IR file given as input to profile IR instructions.	PRO, FR, CG
`noelle-meta-prof-embed`	Embed profiles into the IR file given as input.	PRO, FR, CG
`noelle-meta-pdg-embed`	Compute and embed the PDG into the IR file given as input.	PDG
`noelle-meta-loop-embed`	Assign identificators to loops and their mapping with source code loops.	LS
`noelle-load`	Load the NOELLE abstractions into memory without computing them.	
`noelle-fixedpoint`	Invoke a custom tool until it changes the IR.	
`noelle-codesize`	Print to standard output the number of static IR instructions of the IR file given as input.	
`noelle-loopsize`	Print to standard output the number of static IR instructions of the IR file given as input that are included in loops.	
`noelle-arch`	Generate a file that describes the underlying architecture and its profiles (e.g., core-to-core latencies).	AR
`noelle-linker`	Links IR files together while preservering the semantic of metadata generated by NOELLE's tools.	
`noelle-bin`	Generate a standalone binary from an IR file using the compilation options specified as metadata inside the IR file given as input.	
`noelle-config`	Print to standard output information about the installation of NOELLE (e.g., the installation directory).	

engine that can be used to implement data flow analyses. DFE implements conventional optimizations like bit-vectors, basic block granularity optimization, working list algorithm, and loop-based priority [24]. Finally, NOELLE provides a set of common data flow analyses that rely on DFE.

Profiler (PRO). NOELLE provides several code profilers, the ability to embed their results into IR files, and abstractions to facilitate high-level queries on such data. Examples of queries that can be performed are the hotness of a code region (e.g., a loop, an SCC of a dependence graph), loop-specific information (e.g., loop iteration count, average loop iteration count per invocation), and function-specific information (e.g., the average number of times that an invocation of a function invokes another).

Scheduler (SCD). NOELLE provides the *scheduler* abstraction that offers the capability of moving instructions within and among basic blocks while preserving original code semantics. This abstraction enables users to work at different granularities; users can move a whole loop, a basic block, or single instructions. The scheduler relies on the PDG abstraction to guarantee semantic preservation. The abstraction provides a hierarchy of schedulers starting from a generic one and including loop-specific and within-basic-block schedulers. Each scheduler augments the generic capabilities with specialized capabilities (e.g., reducing the header size of a loop).

Loop Transformer (LT). NOELLE offers the *loop transformer* abstraction that enables passes to modify/create/delete loops. LT is similar to the IRBuilder abstraction offered by LLVM, but instead of targeting instructions, LT targets loops.

Induction variables (IV). NOELLE provides the *induction variable* abstraction. Because LLVM's IR is in SSA form, the concept of the loop's induction variable is embodied by an SCC of the SCCDAG of that loop. NOELLE's abstraction exposes such SCC, the starting and ending value of an induction variable, the step amount per loop iteration, and whether an induction variable controls the number of loop iterations that will be executed. Finally, IV exposes the potential relationship with other induction variables for those that are derived.

While the latest LLVM version provides an induction variable abstraction, this is unfortunately not enough. The main difference between the LLVM's induction variable and NOELLE's version is that NOELLE implements a more robust algorithm to detect induction variables based on both the SCCDAG and SCEV. This enables NOELLE to detect induction variables independent of the target loop shape. Instead, LLVM can only detect induction variables for do-while loops.

Induction Variable Stepper (IVS). A common operation for modern and emerging code transformations is to modify the step of induction variables. For example, loop rotation needs to revert the step value of induction variables. Another example is an advanced DOALL parallelization, which needs to perform chunking between iterations to increase spatial locality. NOELLE's *induction variable stepper* abstraction offers the capability to modify any step value of induction variables of a loop; users need only specify the new step values, and the abstraction modifies the loop accordingly.

Loop Content (LC). This abstraction includes a representation of the loop structure (called LS). The latter is equivalent to the loop abstraction of LLVM. The abstraction LC, instead, adds to LS the loop dependence graph (computed from the

PDG) and the loop-specific instances of the abstractions IV and INV.

Other abstractions. Above, we have described the most important abstractions NOELLE provides. However, NOELLE provides additional abstractions used for simple compilation tasks such as *control equivalence*, *reduction operations*, *extensible metadata* attached to control structures like loops, *SCCDAG partitioner*, *forests(FR), and graphs* designed to restore connections among remaining parts when a node is deleted, *architecture* to describe how logical cores are mapped to physical cores and NUMA nodes, and deterministic *IDs* for instructions, loops, functions, and basic blocks.

Furthermore, NOELLE offers a new implementation of the *loop structure (LS)*, *dominator*, and *scalar evolution* abstractions. These new implementations avoid common bugs caused by LLVM function passes freeing their memory when a module pass invokes them on multiple functions. To avoid this common bug, NOELLE offers implementations of these LLVM abstractions with the property that only their users can free these memory objects.

C. NOELLE's Tools

NOELLE includes tools (Table IV) to help users deploy their compilation tool-chain The most important ones are summarized as follows:

`noelle-whole-IR` generates a single IR file. Merging all bitcode into a single bitcode file is important for the analyses and transformations that span a wide code region (e.g., the whole program). This tool is based on `gllvm`.

`noelle-rm-lc-dependences` modifies an IR program to remove or reduce the impact of loop-carried data dependencies.

`noelle-prof-coverage` profiles IR code using representative program inputs. At the moment, NOELLE includes an instruction profiler, a branch profiler, and a loop profiler.

`noelle-meta-pdg-embed` computes the PDG of an IR file. This tool computes the PDG by invoking many time-consuming and accurate alias analyses that power NOELLE. Then, this tool embeds the computed PDG as metadata into the IR file so that NOELLE can re-construct the requested abstractions without requiring memory analyses.

`noelle-meta-loop-embed` assigns identificators to loops in the IR language and it embeds the mapping between these loops and loops of the source code files. Source code loops are identified by the file name and line number of the first statement of the related loop. This information is used by custom tools built upon NOELLE that need to keep track of the relation between loops in IR and loops in the source code (e.g., loops in C++ files).

`noelle-load` loads the NOELLE layer in memory. Custom tools invoke NOELLE's empowered LLVM pass by using `noelle-load` rather than the LLVM tool `opt`.

`noelle-arch` measures architecture-specific characteristics. At the moment, this tool measures the core-to-core latency and bandwidth. This tool also interacts with the tool `hwloc` [25] to find the number of physical and logical cores of the underlying platform, their mapping, and NUMA nodes.

Algorithm 1: *isInvariant_llvm(Instruction I, Loop L, Dominator DT, AliasAnalysis AA)*

```
Result: Return true if instruction I is an invariant in loop L
/* Simplified logic of LLVM implementation */
for operand in I.getOperands() do
    if operand is defined in L then return False;
end
if isa<LoadInst>(I) then
    for Instruction J in L do
        if getModRef(J, I) != NoMod then return False;
    end
end
if isa<StoreInst>(I) then
    for memory use MU in L do
        // Conservatively ensures no memory
        // use precedes this store
        if not DT.dominates(I, MU) then return False;
    end
    // Ensures no memory def/use would be
    // invalidated by hoisting the store
    M ← AA.getNearestDominatingMemoryAccess(I);
    if M is in L then return False;
end
if call ← dyn_cast<CallInst>(I) then
    if AA.getModRefBehavior(call) != NoMod then return False;
    S ← AA.onlyMemoryAccessesAreArguments(call);
    if not S then return False;
    for Argument A of call do
        for sL in L-¿getSubLoops() do
            for sI in sL do
                if AA.getModRef(A, sI) != NoMod then return False;
            end
        end
    end
end
return True;
```

Algorithm 2: *isInvariant_noelle(Instruction I, Loop L, PDG dg, Stack s)*

```
Result: Return true if instruction I is an invariant in loop L
/* Implementation using high level abstraction PDG
   instead of low level abstractions alias analysis
   and dominators */
if I in s then return False;
s.push(I);
for PDG dependence J to I do
    if J is in L then
        inv ← isInvariant_noelle(J, L, dg, s);
        if not inv then return False;
    end
end
s.pop();
return True;
```

D. Impact of NOELLE's Abstractions

NOELLE's abstractions may depend on each other to simplify design while keeping high precision. We show the impact of building on higher-level NOELLE abstractions by looking at the invariant abstraction (INV) as an example.

Algorithm 1 shows the simplified logic of LLVM's implementation that relies on low-level abstractions to decide whether a given instruction is a loop invariant. First, the algorithm checks if any operand of `I` is defined within loop `L`. If no operands are defined within `L`, it checks the type of the instruction `I`. If `I` is a load instruction, it checks if any other instruction of `L` can modify the same memory location accessed by `I`. If `I` is a store instruction, it checks if any memory use precedes `I` in `L`. If not, it checks no memory invalidation happens if `I` would be hoisted outside the loop. Finally, if `I` is a call instruction, it checks (i) if `I` can modify any memory location, (ii) if the only memory accessed are via arguments to the call, (iii) and if any sub-loop can modify the same memory accessed via arguments by the call `I`.

Algorithm 2 shows NOELLE's implementation that relies on the high-level PDG abstraction. It checks if `I` is currently under analysis (i.e., in the stack `s`). If not, it checks instruction that `I` depends on whether it is outside the loop or a loop invariant. Notice that this algorithm is smaller, simpler, and more precise than Algorithm 1 (Figure 4).

III. Transformations Built Upon NOELLE

This section describes the code transformations built upon NOELLE. Table V summarizes them and their Lines of Code (LoC). Each transformation relies on several of NOELLE's abstractions. Table VI shows the abstractions used by them. It is important to notice that every abstraction is used by more than one custom tool suggesting their wide applicability.

A. Work Re-Implemented Using NOELLE

Next we describe work previously published that we re-implemented using NOELLE.

HELIX parallelizes a loop by distributing its iterations between cores [26]–[28]. Each iteration is sliced into several sequential and parallel segments. Different instances of the same sequential segment run sequentially between the cores while everything else can overlap.

HELIX uses PRO, FR, and LC of NOELLE to identify the most profitable loops to parallelize. HELIX uses the PDG at the SCC granularity and only for the hot loops. HELIX also uses ENV to identify and organize the live-in and live-out of each chosen loop. LT and T abstractions are then used to generate the parallel version of a loop.

HELIX uses SCCDAG, INV, IV, and the RD abstractions to identify the SCCs that need to be executed sequentially. DFE is used to translate SCCs into sequential segments. SCD is then used to reduce the size of each sequential segment and schedule them within the body of each parallelized loop. Moreover, HELIX uses IVS to chunk loop iterations.

DSWP parallelizes a loop by distributing its SCCs between cores [23]. Instances of a given SCC are executed by the same core to create a unidirectional communication between cores. DSWP uses NOELLE's abstractions, similarly to how HELIX does while leveraging DSWP-specific knowledge to select and subsequently parallelize loops.

DOALL parallelizes a loop that has no loop-carried data dependencies by distributing its iterations among cores [29]. DOALL's implementation uses NOELLE's abstractions similarly to the other parallelizing compilers (DSWP and HELIX), the difference being the loop selection process and parts of the parallelized code generation.

Compiler-Based Timing is co-designed with the underlying operating system to inject calls to OS routines [30] into a program. This compiler uses DFE and PRO to implement its specialized data flow analyses. It also uses LC, FR, and LT to handle potentially-infinite loops. Finally, it uses CG to improve the accuracy of its time analyses.

Time-Squeezer generates code optimized for timing speculative micro-architectures [31], [32]. To this end, the compiler needs to decide when to swap the compare operands (and modify its uses), how to change the schedule of instructions, and where to inject instructions that modify the clock period of the underlying architecture. This custom tool uses DFE, LC, and FR to decide where to inject clock-changing instructions. It then uses SCD to optimize the instruction sequence of a code region that uses the same clock period. Finally, it uses ISL and the whole program PDG at the instruction granularity to analyze the compare instructions and their dependencies.

Loop Invariant Code Motion hoists loop invariants outside their loop. It uses FR to hoist invariants from innermost to outermost loops. INV is then used to identify instructions that can be hoisted. Finally, it performs the hoist with LC.

Perspective is a speculative-DOALL parallelization framework that maintains the applicability of speculative techniques while approaching the efficiency of non-speculative ones. This system relies on the PDG at the instruction granularity for each selected loop and their SCCDAG.

B. Work Enabled by NOELLE

We started NOELLE in 2016 to accelerate the implementation of our compilers. Since then, we implemented the following systems upon NOELLE.

CARAT is co-designed with the underlying operating system to replace virtual memory. This compiler injects code to guard IR memory instructions that cannot be proved at compile time to be valid [33], [34]. CARAT relies on the whole program PDG at the instruction granularity, the SCCDAG, and INV to identify the memory instructions that need guards. It then uses DFE and PRO to avoid redundant guards of the same memory location. CARAT also uses LC, LT, and IV to merge guards. Finally, SCD is used to place the guards in the code.

PRVJeeves selects pseudo-random value generators (PRVG) for a randomized program (e.g., Monte Carlo simulations) [35]. To do so, it uses the whole program PDG at the instruction granularity, CG, and DFE to identify the allocations and uses of the PRVGs. Then, PRVJeeves uses PRO to prune the design space (e.g., PRVGs used infrequently are unmodified). Moreover, it uses LC, LT, INV, and IV to identify the uses of a vector of PRVGs. Finally, PRVJeeves uses SCD to place the uses of a PRVG in the code.

CCK is a compiler co-designed with the Nautilus OS [36] to bring the OpenMP stack in the kernel space by only adding a few lines of code in the OS [37].

Dead Function Elimination eliminates functions that cannot be reached by the main function nor by module constructors. To do so, the system relies on the complete CG, ISL, and the whole program PDG at the instruction granularity. It modifies the code to eliminate functions whose address is

TABLE V: Custom tools built upon NOELLE. LoC of tools that are enabled by NOELLE and not available using only LLVM are marked with * and estimated using the LoC of the NOELLE's abstractions used by such tools.

Custom tool	*Description*	*LLVM*	*LLVM + NOELLE*	*Percent reduction*
HELIX	Parallelizing compiler that applies the HELIX code parallelization technique	15453	958	94%
DSWP	Parallelizing compiler that applies the DSWP code parallelization technique	8525	775	91%
DOALL	Parallelizing compiler that applies the DOALL code parallelization technique	5512	321	94%
Compiler-based timing (COOS)	Compiler to inject calls to Operating System routines to replace hardware interrupts	1641	495	70%
Time Squeezer (TIME)	Compiler to optimize compare instructions for timing speculative architectures	510	92	82%
Loop Invariant Code Motion (LICM)	Hoist loop invariants outside their loop	2317	170	93%
Perspective (PERS)	Parallelizing compiler that minimizes speculation and privatization costs	33998	22706	33%
CARAT	Inject memory guards to potentially incorrect memory instructions	*21899	595	97%
PRVJeeves (PRVJ)	Compiler to select the Pseudo Random Value Generators for the program given as input	*17863	456	97%
CCK	OpenMP compiler co-designed with the Nautilus OS to bring the OpenMP stack in kernel space	*51741	18345	65%
Dead Function Elimination (DEAD)	Reduce the number of functions without increasing the total number of IR instructions	*7512	61	99%
	Total	**166971**	**44974**	**73%**

TABLE VI: NOELLE's abstractions are versatile as they are used by several and significantly different custom tools.

Custom tool	**NOELLE's abstractions used**																	
	PDG	SCCDAG	CG	ENV	T	DFE	PRO	SCD	LC	LT	IV	IVS	INV	FR	ISL	RD	AR	LS
HELIX	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
DSWP	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
DOALL	✓	✓		✓	✓		✓		✓	✓	✓	✓	✓	✓		✓	✓	✓
COOS			✓			✓	✓		✓	✓				✓				✓
TIME	✓					✓		✓	✓	✓				✓	✓			✓
LICM									✓	✓			✓	✓				✓
CARAT	✓	✓				✓	✓	✓	✓	✓	✓		✓					✓
PRVJ	✓		✓			✓	✓	✓	✓	✓	✓		✓					✓
PERS	✓	✓																
CCK	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
DEAD	✓		✓												✓			

stored in memory when the compiler can prove such address cannot be used by any indirect call.

IV. Evaluation

This section evaluates NOELLE and the custom tools built upon it. Before presenting the results, we describe our evaluation platform and our evaluation methodology. Our results show that each of NOELLE's abstraction can be used by several significantly different custom tools. Results suggest that NOELLE's implementation of a few abstractions that exist in LLVM are more precise than their LLVM counterparts. Finally, results suggest that we can build a custom tool in a few lines of code that is powerful enough to improve the performance or reduce the binary size compared to the mainline, widely adopted compilers like `clang`.

A. Experimental Setup

We have evaluated NOELLE and 11 custom tools on the platform described next and by following the measurement methodology described here.

Platform. Our evaluation was done on a Dell PowerEdge R730 server with one Intel Xeon E5-2695 v3 Haswell processor running at 2.3GHz. The processor has 12 cores with 2-way hyperthreading, 35MB of last-level cache, and a peak power consumption of 120W. The cores are supported by 256GB of main memory in 16 dual rank RDIMMs at 2133MHz. Turbo Boost was disabled, and no CPU frequency governors were used (i.e., all cores ran at a maximum frequency). The OS used is Red Hat Enterprise Linux Server 8 on kernel 4.18.

NOELLE is available for several versions of LLVM [1] ranging from 5 to the latest (at the time of writing) 12. Different NOELLE versions compiles differently depending on which external libraries are available for that LLVM version. This paper reports the results for NOELLE compiled using LLVM 9 because one of the external alias analysis, SCAF, that NOELLE relies on is at the moment only available for LLVM 9. Hence, NOELLE with LLVM 9 currently provides the most accurate PDG. Finally, all results are generated using NOELLEGym [38], an infrastructure we built to evaluate NOELLE-related tools. While our artifact targets 71 benchmarks from four suites, most results showed next exclude PolyBench benchmarks for lack of space (similar trends are found in PolyBench).

Statistics and convergence. Each data point we show in our evaluation is the median value of 11 runs.

B. Building Upon NOELLE Reduces Source Code

NOELLE simplifies the implementation of code analyses and transformations. Table V compares the implementations of 11 transformations when built upon NOELLE and when implemented only using LLVM abstractions. We agree LoC is not an ideal proxy for measuring the complexity of a system, however it is important to notice that the reduction in LoC obtained by using NOELLE and shown in Table V is significant (73% less code for the 11 custom tools).

NOELLE abstractions are general enough to be useful by many and highly-heterogeneous custom tools. Table VI shows that each abstraction is used by several custom tools. For example, LT is used by nine custom tools out of 11. Moreover, it is important to notice the heterogeneity of these custom tools that use (for example) LT: parallelizing transformations, loop invariant code motion (LICM), code optimizations for timing speculative micro-architecture (TIME), memory guard injector and optimization (CARAT), PRVG selector (PRVJ), and scheduler of OS routines within applications (COOS).

C. NOELLE Abstractions

Next, we compare the subset of NOELLE's abstractions that are also available in LLVM. These abstractions are loop invariants, loop induction variables, and dependencies.

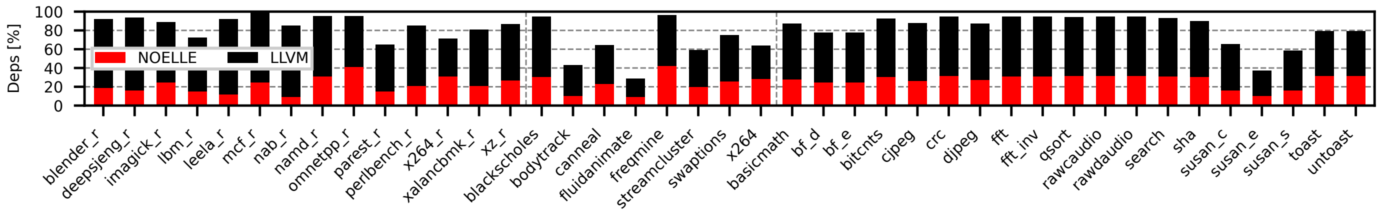

Fig. 3: While LLVM is capable of proving the non-existence of most dependencies, NOELLE disproves significantly more.

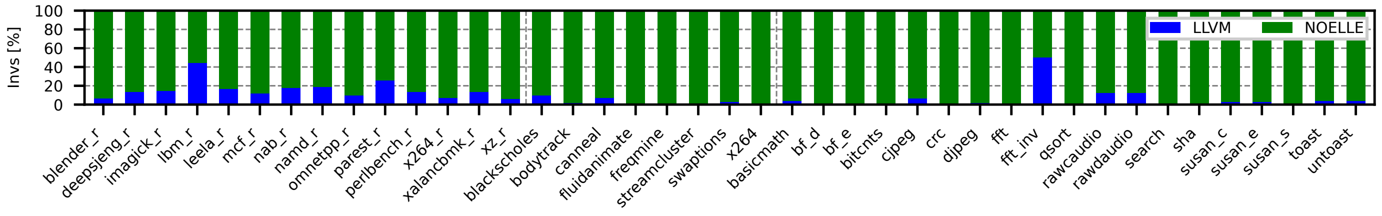

Fig. 4: NOELLE detects significantly more invariants than LLVM even if the former relies on a simpler and shorter algorithm powered by higher-level abstraction (Algorithm 2) compared to LLVM (Algorithm 1).

Figure 3 shows that NOELLE's implementation of dependencies of its PDG is more accurate than LLVM's abstraction. LLVM is capable of proving a significant fraction of potential memory dependencies non-existing. NOELLE further improves this result by leveraging state-of-the-art alias analyses [18], [39], [40]. While theoretically these alias analysis frameworks could be included in LLVM, they are currently designed and implemented outside the LLVM codebase; including them in LLVM will be challenging in practice.

Figure 4 compares the number of loop invariants identified by both LLVM and NOELLE. NOELLE identifies significantly more loop invariants than LLVM because the invariant abstraction of NOELLE is built using the PDG abstraction. This makes the invariant detection algorithm within NOELLE (Algorithm 2) smaller, more elegant, and more powerful compared to the LLVM one (Algorithm 1).

Finally, we computed the number of loop induction variables using both LLVM and NOELLE for the IR generated by `noelle-rm-lc-dependencies`. Among the 71 benchmarks LLVM identifies less loop induction variables (1512 total) compared to NOELLE (3749 total). The reason is that LLVM's induction variable analysis expects the input IR to have loops in the do-while shape. However, most loops in the 71 benchmarks have a while shape, and changing them into a do-while shape would reduce the applicability of all the implemented parallelization techniques (so `noelle-rm-lc-dependencies` keeps loops in their while shape). Instead, NOELLE identifies loop induction variables (3749 total) independently of the shape of a loop.

D. Parallelizing Transformations Upon NOELLE

Next, we describe the parallelizing code transformations built upon NOELLE (HELIX, DSWP, DOALL) that do not rely on speculative techniques. This allows us to compare small code implementations built upon NOELLE with the parallelizing transformations implemented by `icc` and `gcc`.

Figure 5 shows the speedups we obtained in PARSEC and MiBench benchmark suites. The few missing benchmarks have failed to compile with the unmodified `clang` compiler, and therefore we cannot use them to test NOELLE-based tools. Figure 5 shows that the NOELLE-based small custom tools already extract more parallelism compared to what `gcc` and `icc` extract. Furthermore, we analyzed the few benchmarks that NOELLE-based parallelizing tools could not extract significant performance benefits (e.g., `crc`). We found this is due to the lack of support for memory object cloning. This is arguably an abstraction that should exist in the parallelization techniques rather than within NOELLE as the latter is not specialized for parallelization purposes. Finally, it is important to note that Figure 5 shows it is possible to have all these parallelization techniques implemented in the same compiler using the same abstractions (NOELLE is the first codebase that includes both DSWP and HELIX).

We also run these five parallelizing tools on 14 SPEC CPU2017 benchmarks (the only missing benchmark is `gcc`, which did not compile with `gllvm`). Speedups were obtained only by NOELLE-based parallelizing tools and are within 1% and 5% for these 14 benchmarks demonstrating the robustness of NOELLE abstractions. Speculative techniques are likely to be required to unlock further speedups on these benchmarks. We argue that speculative techniques should be implemented outside NOELLE as they are parallelization-specific.

Finally, we have ported a state-of-the-art parallelizing compiler (Perspective [41]) together with the authors. We modified the original codebase to use the PDG and the SCCDAG abstractions. This new version has preserved the performance shown in the authors' original paper.

E. Reducing Binary Size with NOELLE

Binary size is an important optimization goal for both embedded systems and servers [42]. The compiler `clang` offers an optimization level for this goal (`-Oz`). DeadFunctionElimination further reduces the binary size by 2.3% on

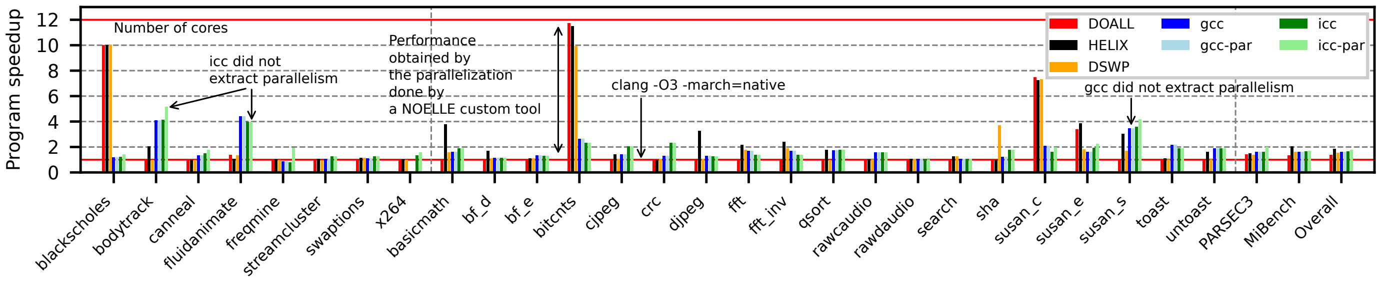

Fig. 5: Both `gcc` and `icc` did not obtain additional performance benefits from their parallelization techniques. Instead, NOELLE-based parallelizing tools generate additional benefits compared to their baseline, `clang`. Finally, both `HELIX` and `DSWP` include the DOALL parallelization to parallelize only loops with no loop-carried dependences.

average among the 71 benchmarks considered. The reduction is obtained by inlining functions that are invoked by a single call instruction; after inlining, these functions are removed.

V. Related Work

A. Providing High-Level Abstractions

Researchers have explored bringing high-level abstractions to compilers in many different ways. Compilers that support automatic parallelization include Polaris [8], a parallelizing compiler for Fortran programs, Cetus [9], a C compiler focusing on multicore, and ROSE [7], a compiler for building custom compilation tools. These compilers operate on high-level abstractions to perform *source-to-source translation*, thus missing opportunities presented only in low-level IRs including fine-grained operations and more canonical code patterns.

Many domain-specific projects add new abstractions similar to NOELLE. SeaHorn [43] provides new abstractions for developing verification techniques. Polly [44], [45], PLUTO [46], HALIDE [11], [47], Tiramisu [48], [49], and APOLLO [50] provide abstractions to suit polyhedral optimizations, targeting loops characterized by regular control and data flows. TensorFlow [10] uses high-level graph representations to discover more graph optimizations [51]. These projects focus on specific domains and their abstractions are not easily reusable for problems outside their domains.

Few domain-independent compilers combine low-level IR with high-level abstractions like NOELLE. SUIF compiler [12] provides low-level IR as well as higher-level constructs including loops, conditional statements, and array accesses [13]. The IMPACT compiler [14] provides hierarchical IRs. Unfortunately, they are not maintained anymore.

The LLVM community also has a Loop Optimization Working Group [52] that recently has started working on a few abstractions included in NOELLE, such as the dependence graph. We plan to share NOELLE code with them. We also see value in maintaining NOELLE as a separate project that focuses mainly on performance rather than making a balance between performance, and compilation time.

B. LLVM Projects

As we have built NOELLE on top of LLVM, we want to know how NOELLE might impact compiler research. To do this, we have exhaustively reviewed all 544 papers published in PLDI, CGO, and CC during the past five years (2016-2020). Out of these papers, 87 papers explicitly mention that they are built on top of LLVM by either implementing new passes, modifying the LLVM internals, or creating a new front-end/back-end based on LLVM IR. Out of these 87 papers:

- 26 (29.9%) use abstractions similar to those provided by NOELLE. Thus, they could be re-implemented on top of NOELLE with significantly fewer lines of code and/or better performance. Of these, we have implemented CARAT [33] and PRVJeeves [35] with NOELLE and presented the benefits in Section III. Other examples include Spinal Node [53], which uses PDG as well as data flow analysis; Valence [54], which relies on call graph analysis; Clairvoyance [55], which relies on loop-carried dependence analysis.
- 10 (11.5%) provide new abstractions or implement analyses or transformations that fulfill NOELLE abstractions. We have already integrated SVF [19] and SCAF [18] within NOELLE. We plan to evaluate others [56]–[59] in the future.
- 25 (28.7%) are doing tasks orthogonal to NOELLE's abstractions. Nevertheless, they do not conflict with NOELLE because both implementations do not modify LLVM internals. Due to NOELLE's modular and demand-driven design, future work can use NOELLE even if only a subset of abstractions are of interest.
- 26 (29.9%) papers modify LLVM internals or use alternative front/back-ends. These projects need to be analyzed case by case for the possibility of integration with NOELLE.

In conclusion, 41.4% of the projects are highly likely to benefit from or contribute to NOELLE's abstractions; 28.7% have the potential for future collaboration; 29.9% need investigation before integration.

VI. Conclusion

Code analyses and transformations need to go beyond peephole and ILP optimizations for modern architectures. Their implementation requires high-level abstractions that are currently lacking in LLVM. This paper introduces NOELLE, an open-source compilation layer built upon LLVM that provides the required abstractions. NOELLE has been tested with 11 highly diverse tools that are built upon it. All of these tools gain benefits compared to unmodified LLVM.

APPENDIX

A. Abstract

This artifact describes the tools and code used in our evaluation. The main component is a docker image which includes a detailed README, scripts to generate and run all experiments along with the LLVM9.0.0 already installed. It requires docker to run the image and a network connection to pull down external dependencies and our benchmark suite. Evaluating this artifact requires an Intel multicore processor with shared memory. The scripts will generate all of the data from our paper evaluation into a text format with the only manual step required being to add the SPEC CPU2017 benchmark suite as we cannot share it directly. The script will optionally generate the SPEC CPU2017 data by following the instructions in `README.md`

B. Artifact Check-List (Meta-information)

- **Algorithm: No**
- **Program: PARSEC3, PolyBench, MiBench**
- **Compilation: LLVM9.0.0, Included**
- **Transformations: None**
- **Binary: None**
- **Data set: Data sets are included with the benchmark suite**
- **Run-time environment: None**
- **Hardware: None**
- **Run-time state: Yes**
- **Execution: Sole user, Pinning, approximately 4 days to run**
- **Metrics: Execution time, Number of dependences, Number of induction variables, Number of loop invariants**
- **Output: Individual file output for each metric and each benchmark**
- **Experiments: The experiments can be run with the included** `bin/compileAndRun` **script. The user must set environment variables to customize the experiments as described in the README.md include in the docker image.**
- **How much disk space required (approximately)?: 500 GB**
- **How much time is needed to prepare workflow (approximately)?: Several hours**
- **How much time is needed to complete experiments (approximately)?: 4 days**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Data licenses (if publicly available)?: No**
- **Workflow framework used?: Customization and automation of the experiments are handled by the** `bin/compileAndRun` **script and environment variables described in the README.md included in the docker image.**
- **Archived (provide DOI)?: 10.5281/zenodo.5789400**

C. Description

1) How Delivered: The artifact can be accessed via a docker image following the DOI.

2) Hardware Dependencies: Intel multicore chip with shared memory required. TurboBoost and HyperThreading must be disabled to reproduce execution time results.

3) Software Dependencies: Running the artifact requires `docker`, all other software dependencies are included in the docker image or installed when running the included scripts. The only dependency that must be added is the SPEC2017 benchmark suite which cannot be included.

4) Data Sets: All data sets will be automatically downloaded when the scripts within the container are run. The SPEC CPU2017 data set cannot be included in the artifact because of licensing, if the reviewer would like to obtain SPEC CPU2017 results, they must add it themselves.

D. Installation

After downloading the docker image, add it to your docker with `docker load < noelle.tar`. After adding the image to your docker it can be run interactively with `docker run --rm -it noelle /bin/bash`

```
docker load < noelle.tar
docker run --rm -it noelle /bin/bash
```

E. Experiment Workflow

The workflow for this experiment is as follows.

1) NOELLE is compiled.
2) All benchmarks are compiled for all configurations (WARNING: this will take several hours).
3) Statistics are generated about loops (induction variables, invariants) and dependencies in the PDG of a program.
4) Baseline times are generated for all benchmarks.
5) Times are generated for binaries obtained by `noelle-rm-lc-dependencies`, which is the input of the parallelization schemes. This configuration is called NONE.
6) Times are generated for DOALL parallelized binaries for all benchmarks.
7) Times are generated for HELIX parallelized binaries for all benchmarks.
8) Times are generated for DSWP parallelized binaries for all benchmarks.
9) The speedups are computed.
10) Statistics about how many loops have been parallelized with which techniques are generated.

This workflow is automatically run through the `bin/compileAndRun` script. This can be launched in the background in order to watch the progress at a finer grain through the output.txt file.

After running the docker image interactively, please read README.md

```
vim README.md
```

Launch in the background building NOELLE and running the experiments:

```
./bin/compileAndRun &
```

Optional: View additional progress of the script

```
tail -f output.txt
```

F. Evaluation and Expected Result

After running the docker image interactively, the README.md includes instructions for the evaluator to run the experiment. This is provided as a single script that takes no arguments, run from the home directory of the docker image. Generating results for all benchmarks takes approximately 4 days. The results directory includes the author's results that were used in the paper submission. It is expected that the results generated by the artifact are in line with the author's results.

G. Experiment Customization

There are three experimental configurations and one customization available in this artifact.

- **Minimal** runs the minimal set of experiments (PARSEC3, MiBench) to support the claims made in the submitted paper. This experimental configuration takes approximately 4 days to run.
- **Submission** runs the SPEC CPU2017 benchmark suite, which are included in the submitted paper. This experimental configuration takes approximately 12 days to run. Due to the long experimental time, this configuration is separate from the **Minimal** set. This can be selected by setting the `NOELLE_SUBMISSION` environment variable.
- **Final** runs new results that were not included in the paper submission, but will be included in the final version of the paper. This experimental configuration takes approximately 5 days to run. This can be selected by setting the `NOELLE_FINAL` environment variable.
- **Runs** changes the number of times that time-sensitive evaluations are run. The default number of runs is 5. This can be modified with the `NOELLE_RUNS` environment variable.

H. Notes

For more detailed information about the artifact and its evaluation process, read the README.md located in the artifact, which has been made publicly available.

I. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[2] "The LLVM Compiler Infrastructure Project," http://llvm.org/.

[3] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)," https://gcc.gnu.org/.

[4] "MSVC auto-parallelization," https://docs.microsoft.com/en-us/cpp/parallel/auto-parallelization-and-auto-vectorization?view=vs-2019.

[5] "Automatic Parallelization," https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/automatic-parallelization.html.

[6] "AutoParInGCC - GCC Wiki," https://gcc.gnu.org/wiki/AutoParInGCC.

[7] "Rose Compiler – Program Analysis and Transformation," http://rosecompiler.org/.

[8] B. Blume, R. Eigenmann, K. Faigin, and J. Grout, "Polaris: The Next Generation in Parallelizing Compilers," Tech. Rep.

[9] "The Cetus Project," https://engineering.purdue.edu/Cetus/.

[10] "TensorFlow," https://www.tensorflow.org/.

[11] "Halide," https://halide-lang.org/.

[12] "The SUIF Compiler - SUIF 2," https://suif.stanford.edu/suif/suif2/.

[13] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler," Tech. Rep.

[14] P. P. Chong, S. A. Mohike, and N. J. Warier, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," p. 10.

[15] "The NOELLE Project," https://github.com/scampanoni/noelle.

[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[17] A. W. Appel, *Modern compiler implementation in C*. Cambridge university press, 2004.

[18] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August, "Scaf: A speculation-aware collaborative dependence analysis framework," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 638–654. [Online]. Available: https://doi.org/10.1145/3385412.3386028

[19] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 265–266.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[21] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 432–447. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173181

[22] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[23] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, Nov 2005, pp. 12 pp.–118.

[24] A. W. Appel, *Modern compiler implementation in Java*. Cambridge university press, 2008.

[25] "Portable hardware locality (hwloc)," https://www.open-mpi.org/projects/hwloc.

[26] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259028

[27] N. Murphy, T. Jones, R. Mullins, and S. Campanoni, "Performance implications of transient loop-carried data dependences in automatically parallelized loops," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 23–33. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892214

[28] S. Campanoni, T. Jones, G. Holloway, G. Y. Wei, and D. Brooks, "The helix project: Overview and directions," in *DAC Design Automation Conference 2012*, June 2012, pp. 277–282.

[29] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee, "Parallelization of doall and doacross loops—a survey," in *Advances in computers*. Elsevier, 1997, vol. 45, pp. 53–103.

[30] S. Ghosh, M. Cuevas, S. Campanoni, and P. Dinda, "Compiler-based timing for extremely fine-grain preemptive parallelism," in *Super Computing conference (SC)*, 2020.

[31] Y. Fan, S. Campanoni, and R. Joseph, "Time squeezing for tiny devices," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 657–670. [Online]. Available: https://doi.org/10.1145/3307650.3322268

[32] Y. Fan, T. Jia, J. Gu, S. Campanoni, and R. Joseph, "Compiler-guided instruction-level clock scheduling for timing speculative processors," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: ACM, 2018, pp. 40:1–40:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3196013

[33] B. Suchy, S. Campanoni, N. Hardavellas, and P. Dinda, "CARAT: A case for virtual memory through compiler- and runtime-based address translation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 329–345. [Online]. Available: https://doi.org/10.1145/3385412.3385987

[34] B. Suchy, S. Ghosh, A. Nelson, Z. Huang, D. Kersnar, S. Chai, M. Cuevas, A. Bernat, G. Chaudhary, N. Hardavellas, S. Campanoni, and P. Dinda, "CARAT CAKE: Replacing paging via compiler/kernel cooperation," in *ASPLOS*, 2022.

[35] M. Leonard and S. Campanoni, "Introducing the pseudorandom value generator selection in the compilation toolchain," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 256–267. [Online]. Available: https://doi.org/10.1145/3368826.3377906

[36] K. C. Hale and P. A. Dinda, "Enabling hybrid parallel runtimes through kernel and virtualization support," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 161–175, 2016.

[37] J. Ma, W. Wang, A. Nelson, M. Cuevas, B. Homerding, C. Liu, Z. Huang, S. Campanoni, K. Hale, and P. A. Dinda, "Paths to openmp in the kernel," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[38] "NOELLEGym: where NOELLE-based tools exercise," https://github.com/scampanoni/noelleGym.

[39] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August, "A collaborative dependence analysis framework," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 148–159. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049849

[40] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[41] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 351–367. [Online]. Available: https://doi.org/10.1145/3373376.3378458

[42] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 462–473. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[43] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn Verification Framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, vol. 9206, pp. 343–361.

[44] "Polly - Polyhedral optimizations for LLVM," https://polly.llvm.org/.

[45] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[46] U. Bondhugula and J. Ramanujam, "Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer," 2007.

[47] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 519–530.

[48] "Tiramisu Compiler," http://tiramisu-compiler.org/.

[49] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, M. T. Kandemir, A. Jimborean, and T. Moseley, Eds. IEEE, 2019, pp. 193–205.

[50] J. M. M. Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative POLyhedral Loop Optimizer," in *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*, Jan. 2017, p. 8.

[51] "TensorFlow graph optimization with Grappler — TensorFlow Core," https://www.tensorflow.org/guide/graph_optimization.

[52] llvm, "The Loop Optimization Working Group," https://llvm.org/devmtg/2019-10/talk-abstracts.html#pan2.

[53] B. Kim, S. Heo, G. Lee, S. Song, J. Kim, and H. Kim, "Spinal code: Automatic code extraction for near-user computation in fogs," in *Proceedings of the 28th International Conference on Compiler Construction - CC 2019*. Washington, DC, USA: ACM Press, 2019, pp. 87–98.

[54] T. Zhou, M. R. Jantz, P. A. Kulkarni, K. A. Doshi, and V. Sarkar, "Valence: Variable length calling context encoding," in *Proceedings of the 28th International Conference on Compiler Construction - CC 2019*. Washington, DC, USA: ACM Press, 2019, pp. 147–158.

[55] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, V. J. Reddi, A. Smith, and L. Tang, Eds. ACM, 2017, pp. 171–184.

[56] J. Doerfert, T. Grosser, and S. Hack, "Optimistic loop optimization," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, V. J. Reddi, A. Smith, and L. Tang, Eds. ACM, 2017, pp. 292–304.

[57] S. Manilov, C. Vasiladiotis, and B. Franke, "Generalized profile-guided iterator recognition," in *Proceedings of the 27th International Conference on Compiler Construction - CC 2018*. Vienna, Austria: ACM Press, 2018, pp. 185–195.

[58] M. Maalej, V. Paisante, P. Ramos, L. Gonnord, and F. M. Q. Pereira, "Pointer disambiguation via strict inequalities," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, V. J. Reddi, A. Smith, and L. Tang, Eds. ACM, 2017, pp. 134–147.

[59] A. Phulia, V. Bhagee, and S. Bansal, "OOElala: Order-of-evaluation based alias analysis for compiler optimization," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 839–853.