Modern LLVM-based Compiler Autotuning for WCET Optimization

Gabriele Magnani, Davide Baroffio, Federico Reghenzani, Giovanni Agosta, William Fornaciari
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
name.surname@polimi.it

***Abstract*—The problem of compiler optimization selection and ordering, known in the literature as *compiler autotuning*, has been tackled many times for average-case execution time reduction. Optimizing the WCET is becoming a prominent problem for modern hard real-time systems, where the difficulties in accurate WCET estimation hinder the full exploitation of computing platform capabilities. In this article, we propose a novel methodology and a tool based on LLVM for iterative WCET-driven compiler autotuning, which is the first strategy to operate at function-level granularity and to consider not only the selection of optimization passes, but also their ordering. Our findings show that standard optimization levels O0, O1, O2, and O3 are suboptimal when targeting the WCET, and that a per-function selection and ordering of the transformations is necessary. Experimental results show that our approach outperforms the standard optimizations and opens up new directions for future research.**

***Index Terms*—Compilers, Real-Time Systems, WCET, Static Timing Analysis**

I. Introduction

Estimating the Worst-Case Execution Time (WCET) is a critical step in the design and implementation of real-time systems, since the complexity of modern processor architectures and software tasks prevents a precise estimation of a program's timing boundaries. As a result, platform and software model approximations are often necessary, leading to *untight* [1] WCET estimations. The evolution towards heterogeneous architectures and more complex software exacerbates the WCET estimation problem.

The WCET metric has a direct impact on system schedulability because it affects the total theoretical utilization of the system. System utilization can be optimized in two ways: 1) improving the WCET estimation process, or 2) reducing the actual WCET required by the task. In this article, rather than focusing on improving the WCET estimation methodology, we focused on the effects of compiler optimizations on the WCET value and WCET estimations.

Modern compilers apply several optimizations to the code. For instance, the LLVM compiler, when configured with the `-O2` option, applies a total of 201 optimizations (LLVM version 19.1.7, Jan. 2025). Such optimizations have been selected by the compiler developers to include optimizations that, in most scenarios, reduce the average execution time. In this work, we investigate whether these optimizations improve the WCET, rather than only the average execution time.

A. Motivational examples

Let us consider three common compiler optimizations:

1) *Constant folding and constant propagation*: The compiler computes the constant expressions, i.e., expressions whose operands are known at compile-time, and propagates them, substituting their known value in all the expressions that use them.
2) *Trace scheduling*: After statically profiling the application via branch prediction, the compiler performs instruction reordering so that the paths with the highest probability of being executed run faster, at the expense of the less probable paths.
3) *Loop unrolling*: The compiler expands loops by eliminating the jumps and other instructions related to the loop counter. These actions aim at reducing the overhead given by the presence of a loop, increasing binary size.

These three optimizations are often enabled in modern compilers because they increase the code's average performance. However, let us focus on the effects on the WCET: optimization (1) likely improves also the WCET, as constant folding and expansion reduce the execution time; optimization (2) likely worsens the WCET, because the less probable paths (including the worst-case path) are *deoptimized* to improve the average case; optimization (3) has non-clear effects on the WCET because it risks inflating the number of instructions and data cache misses, increasing register pressure. These examples show how the problem of investigating the effect of compiler optimizations on the WCET is non-trivial and, due to the increasing number of available compiler optimizations, is infeasible to conduct manually.

B. State of the Art

Compiler autotuning is the family of methodologies that aims at solving the problem of finding the "best" sequence of optimizations to apply to a program. Although compilers propose predefined optimization sequences – typically mapped to optimization levels, like `-O1`, `-O2`, etc. – that are expected to achieve good average performance improvements, they may not provide the best possible optimization. Because the compilation time is in the order of magnitude of one second or larger, and the exploration space is made of thousands of possibilities, compiler autotuning often rely on finding approximate solutions. Ashouri et al. [2] surveyed scientific papers that exploit machine learning to solve the problem.

The more traditional research on WCET-driven optimizations studies new transformations with the objective of reducing the WCET. This family of WCET-oriented code trans-

formations optimizes the code to reduce the impact of the cache, branch predictors, and loops on the WCET, effectively tackling most of the issues described in Section II-B. For instance, some previous works applied specific optimizations to the worst-case path [3], [4]. Falk et al. [5] integrated the aiT WCET analyzer in a custom compiler with the goal of designing WCET-aware optimizations. Khelassi et al. [6] investigated the effect of optimization levels on execution time variability. A different perspective of the problem has also been investigated [7]: how to select optimizations that improve the efficiency of the downstream WCET analysis.

Compiler autotuning methods for the average case spurred the development of similar approaches for WCET, even if this research area is still immature and open. The general idea is to exploit pre-existing mainstream compiler optimizations to automatically select the sequence of passes that optimize the code with respect to the WCET. The research by Becker and Chakraborty [8] uses statically-derived profiling information to instrument the compiler to optimize only the critical path of the program. The work of Dardaillon et al. [9], instead, performs iterative improvements to the WCET leveraging the output of the commercial aiT static analyzer. The exploration is guided by a heuristic that exploits association rules characterization of optimization passes.

C. Contributions

This paper highlights the challenges related to generating compiler optimization sequences with WCET as a target. To solve them, it proposes a novel methodology with the associated tool, named *Lavinium*, based on the LLVM compiler toolchain and the LLVMTA open-source static analyzer:

- Lavinium is integrated in LLVM and, as a result, gains all the advantages offered by the LLVM framework.
- Lavinium leverages the LLVMTA tool for the estimation of the WCET during compile-time. Having the WCET information in the compilation process has significant advantages, as also suggested by previous works [10].
- LLVM is enhanced with a compilation flow that allows rescheduling optimizations after part of the backend is executed to perform the timing analysis. In this way, the overall compilation time is notably reduced compared to a full compilation for each optimization sequence.
- Our methodology provides a method to *select* and *order* compiler transformations to optimize the WCET at the function-level. State-of-the-art approaches, instead, either focused on average-case optimizations or on WCET optimizations, but without considering the ordering and acting at higher levels. This article presents the first compiler autotuning strategy for WCET optimization that operates at the function-level granularity and considers not only the *selection* of optimization passes, but also their *ordering*.

The proposed approach is then subject to experimental evaluations via three main experimental campaigns. First, the validation of the estimated WCET on several benchmarks, to check that it is consistent with the results of a state-of-the-art tool. Second, the assessment of the WCET improvement given by our methodology compared to standard optimizations. Third, the assessment of the WCET improvement and the empirical validation of the WCET. The code is run on a modern RISC-V board to assess the optimization effects and show that the optimization sequences provided by Lavinium reduce the worst-case observed time on real hardware.

II. Background

Before presenting our methodology, this section provides an overview of the LLVM compiler framework and the state-of-the-art WCET static estimators, with a special focus on the LLVMTA [11] timing analyzer, exploited by our tool.

A. Structure of LLVM

LLVM [12] is a multi-stage compilation framework that converts high-level source code into executable machine code. This process is delineated into three principal phases: the front-end, the middle-end, and the back-end. The front-end is responsible for semantic analysis, syntax analysis, and translation of the source code into a target-independent intermediate representation. Each programming language has its own frontend; for example, C uses *clang* while Rust uses *rustc*. The middle-end (*OPT*) and back-end (*LLC*) focus on optimizations and the generation of machine-specific code. The optimization process utilizes the LLVM Intermediate Representation (LLVM-IR), which provides a high-level program abstraction. LLVM-IR is designed to be agnostic to specific hardware architectures and source languages, thus facilitating a range of optimizations applicable across various platforms. At this stage, various transformations are conducted to enhance performance and minimize code size while preserving the source code's original semantics. Each of these stages can be invoked separately, while the frontend programs are as many as the supported languages.

1) LLVM-IR: The LLVM-IR is the level at which most compiler optimizations, named passes, which can be either analysis or transformation passes. Analysis passes examine the intermediate representation to gather essential information, such as control flow, data dependencies, and potential aliasing issues. Transformation passes, on the other hand, take the information provided by analysis passes and use it to directly modify the IR. In this paper, we analyze the impact of IR transformation passes on the WCET.

2) Back-end transformations: While the middle-end presents a magnitude of different passes that the end user can manually select, the back-end transforms optimized LLVM IR into efficient target-specific machine code through a series of rigidly fixed interrelated phases. Register allocation is at the heart of transitioning from an abstract representation to a concrete hardware implementation. Here, the infinite space of virtual registers is mapped onto the finite set of target registers. With the register allocation completed and stack space requirements determined, the compiler injects prolog and epilog code for the functions. The compilation terminates with the code emission phase, where the fully optimized

target-tailored machine code is output in either an assembly language or a binary format, ready for execution.

3) Pass managers: A pass manager is a core component of LLVM in charge of scheduling transformation passes and analyses to be run on IR in a specific order. LLVM's pass managers are designed to operate at different levels of granularity within the IR, with specialized passes for each level. When a pass manager operates at a specific granularity, it cannot modify the IR at a lower level. The Module granularity is the most general level, allowing passes to alter the entire program as a unit. At this level, passes can refer to function bodies in no particular order and add or remove functions. Function granularity allows passes to operate on each function independently without considering other functions in the program. Similarly, loop granularity enables passes to work on each loop within a function independently from other loops. Passes at this level process loops in a nested order, meaning the outermost loop is processed last. Moreover, LLVM currently employs two pass managers: the legacy pass manager and the new pass manager. As it stands, the new pass manager is applied only to the middle-end optimization pipeline that works with LLVM IR. The backend code generation pipeline relies solely on the legacy pass manager, primarily because most code generation passes operate on Machine IR (MIR) rather than LLVM IR.

B. WCET timing analyses

Timing analyzers have the task of determining the upper bound on the execution time of real-time tasks when executed on particular hardware. There are many challenges that need to be tackled when dealing with WCET estimations, which can be attributed to three main factors:

1) The execution flow is typically data-dependent, meaning that the sequence of instructions being executed depends on the input data, as different data may lead to different paths in the CFG. Finding the worst-case execution path is typically not a trivial task.
2) The hardware state may affect the execution time of a sequence of instructions. In fact, due to microarchitectural features of modern processors – e.g., caches and pipelines – the execution time of the same set of instructions may greatly vary depending on the state of the processor.
3) Timing anomalies prevent analyzers from greedily searching for the upper bound by seeking the worst-case for each instruction. Without diving into the details of timing anomalies [13] – which is a very complex problem outside the scope of this work – we just need to consider that they heavily impact the applicability and the tightness of WCET analyses, which may be required to foresee the execution several steps deep before selecting the worst-case.

WCET analyzers can be classified into two different families: static methods and measurement-based methods. Measurement-based methods do not typically guarantee provable upper bounds for the WCET unless the software and hardware architecture analyzed are rather simple. For this reason and because we need WCET information at compile-time to drive optimization selection, we only focus on static WCET analyzers. For a more detailed description of the techniques employed in WCET analysis, please refer to the work of Wilhelm et al. [14], who survey different methods for solving the WCET problem.

C. The LLVMTA timing analyzer

The static WCET analyzer of choice for Lavinium is LLVMTA from Hahn et al. [11]. LLVMTA is an open-source LLVM-based WCET analysis tool developed at Saarland University. Initially designed to study WCET for architectures with timing anomalies [13], LLVMTA implements state-of-the-art features that, to the best of our knowledge, are only available in the commercial aiT analyzer from AbsInt GmbH [15]. In fact, LLVMTA improves other academic WCET estimation tools [16]–[18] as it exploits *abstract execution graphs* [19], supports compositional analysis [20], and is applicable at the LLVM Intermediate Representation (IR) and Machine-IR (MIR) level, making it suitable to integrate within the LLVM compilation flow.

The LLVMTA's analysis is performed during the execution of the LLVM code generator. During this phase, most of the code has been lowered to MIR, which is very close to assembly language. In this way, LLVMTA is able to infer the memory layout and instruction addresses, also needed for cache modeling. Notably, since the analysis is performed before linking (object files have not been emitted yet), functions from external compilation units cannot be accessed, and their WCET cannot be assessed. To tackle this issue, LLVMTA requires the user to manually instruct the compiler on the expected address of external functions and their WCET.

The analysis performed by LLVMTA can be divided into three main steps:

- Analyze the possible execution paths of the program, including the maximum number of times a loop can be executed (loop bounds). LLVMTA computes loop bounds via the LLVM Scalar Evolution (SCEV) analysis. If SCEV fails, LLVMTA emits a configuration file to hand-fill to associate loop bounds with each program loop.
- Determine the execution times of the program paths. This analysis must account for microarchitectural features since they impact the time (clock cycles) required to execute specific instructions. LLVMTA performs a cycle-by-cycle simulation of the microarchitectural state, constructing an *Abstract Execution Graph* (AEG). User-defined parameters can configure the target abstract architectural model to mimic specific target hardware features.
- Finally, LLVMTA converges the information from the previous two steps and creates a set of constraints and defines an objective function to maximize. This renders the problem of finding the worst-case execution time solvable via an Integer Linear Program (ILP) solver, which outputs the WCET estimate, in clock cycles.

III. Lavinium Design

Lavinium is an LLVM-based all-in-one WCET analyzer and optimizer that performs optimization selection and ordering to reduce the WCET of the program. Relying on LLVMTA, the WCET bounds of Lavinium are computed on the chosen LLVMTA abstract MIR-level hardware model, and not on real hardware. The design and the verification of the static analyzer are outside of the scope of this work: Lavinium embeds LLVMTA and only uses it to guide the exploration. Given an input source file, Lavinium compiles it into a binary guided by the output of LLVMTA, so that each function is optimized to reduce the WCET of the whole program on the LLVMTA abstract model, also providing in the output the estimated optimized WCET in clock cycles.

One of the novelties of our work is that the decision of which transformations to apply is not made at the compilation unit level. Among other state-of-the-art works [8], [9], [21], only the framework proposed by Puaut et al. [21] avoids compilation unit-level optimization exploration. Their approach relies on loop *outlining*, which isolates code snippets (loop bodies) into separate functions. However, this method introduces unnecessary timing penalties due to the need to add extra function calls and parameter passing that may be detrimental to the WCET. Moreover, outlining requires placing the functions into different compilation units, so that they can apply LLVM's `opt` to each function separately, in fact regressing once again into unit-level optimization exploration. In contrast, Lavinium, built within the LLVM framework, explores the optimization space by directly orchestrating the pass manager to apply different sequences for each function in the compilation unit. With this innovative approach, Lavinium explores each function's optimum separately while avoiding unnecessary overheads and reducing compilation errors due to code instrumentation. Therefore, Lavinium can tailor optimization sequences for each function, improving the final optimization result. The timing analysis is performed iteratively on the entire compilation unit by exploiting the LLVMTA open-source static analyzer previously described in Section II-C. For this work, we ported LLVMTA to LLVM version 17 so that it could be seamlessly integrated with Lavinium.

The Lavinium compilation and optimization flow is split into two loops. The inner one, described in Section III-A, performs the design space exploration for a single function, while the outer loop, described in Section III-B, iteratively updates the IR for each function.

A. The Lavinium inner loop

The inner loop of Lavinium is outlined in Figure 1, and we refer to it as `LaviniumOptimize` in the subsequent paragraphs. The procedure takes in input an LLVM-IR and a function, and returns a version of the original IR in which the chosen function is optimized according to the policy selected. First, `LaviniumOptimize` receives the IR (1) and performs a copy of it (2). The copied IR is, in the initial round, immediately fed to LLC to generate the MIR.

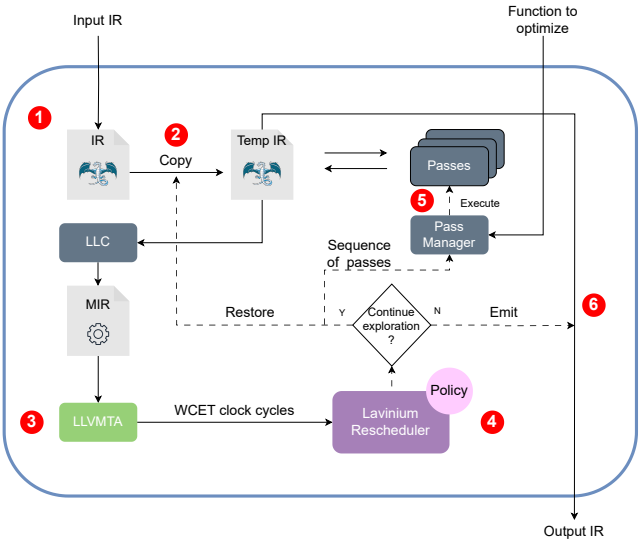


Fig. 1. Outline of the Lavinium optimization flow (`LaviniumOptimize`).

Please note that we do not actually call LLC, but we instead run the CodeGen machine passes (stopping after running the `PatchableFunction` pass). In this way, MIR and all the backend information about registers and instructions are available to be fed to LLVMTA (3), which performs the timing analysis. Thanks to this design, LLVMTA's analysis is performed in the backend, after all IR-level passes and most backend-level passes have been applied, immediately before the assembly emission phase. Thus, no object code or assembly is generated in these phases. LLVMTA computes its estimation of the WCET (in clock cycles) that is then provided to the *Lavinium Rescheduler* (4). This component is the core of the Lavinium approach. The *Lavinium Rescheduler* has four tasks: it prepares the internals of LLVM to perform a new optimization (not depicted), it triggers a copy of the original IR, it runs the policy, and it decides on whether to continue the exploration or not. In the first round, no optimizations have been performed yet, and, therefore, the *Lavinium Rescheduler* just runs the policy for the first time, providing the sequence of transformation passes to execute (which pass and their ordering) to the LLVM Pass Manager. The policies are described later in Section IV. The LLVM Pass Manager then triggers the optimization of the input function on the copied IR (5). After the first sequence of transformations is performed, the new IR is then fed again into the loop via LLC, LLVMTA (3), and then again into the *Lavinium Rescheduler* (4). The *Lavinium Rescheduler* runs the policy and makes the decision. If the decision is to continue the exploration, then the new sequence of transformations is provided to the Pass Manager (5) and a new loop restarts. Conversely, if the *Lavinium Rescheduler* declares the optimization completed, it triggers the emission of the optimized IR (6). Please note that the optimizations applied to this output IR are applied only to the specific function provided as input to `LaviniumOptimize`.

The applied transformations would sometimes eliminate

Algorithm 1 Pseudocode of the outer loop

Input: Source Code S
Output: Compiled Binary B
1: $IR \leftarrow$ `frontend`(S)
2: **for each** function F in IR **do**
3: $IR \leftarrow$ `LaviniumOptimize`(IR, F) ▷ Iterative update of the IR
4: $A \leftarrow$ `LLC`(IR) ▷ Full execution of the backend
5: $B \leftarrow$ `assembler`(A)

loop-bound annotations from the IR, which are accessed by LLVMTA when SCEV fails to estimate a loop's bound. In Lavinium, every instruction in a loop's basic block is annotated with LLVM metadata, which mitigates the effects of instruction reordering or elimination. In our experiments, this mitigation proved to be effective in all cases.

B. The Lavinium outer loop

Lavinium decides which *function*-based and *loop*-based optimization passes to execute in the `LaviniumOptimize` loop for each function. Notice that, although the optimizations are applied at the function level, LLVMTA performs the WCET analysis on the entire MIR, including all functions, each time. Thus, Lavinium does not need to track cache state or other cross-function information separately, since it runs LLVMTA from scratch at every iteration. The overall compilation flow is instead outlined in the pseudocode of Algorithm 1. The source code of the single compilation unit is compiled into the IR by the front-end (Line 1). For each function (Line 2), the `LaviniumOptimize` loop is called and the IR is iteratively updated (Line 3). Eventually, the final IR is compiled by the back-end and emitted with the assembler (Lines 4-5). The final emitted code consists of all functions, each optimized with the optimization sequence that yields the lowest WCET for that function.

1) Function-based iterative update: To the best of our knowledge, this is one of the first attempts to perform such exploration per function rather than considering the overall compilation unit. This novel compilation flow has important advantages. Prior work, such as the framework by Puaut et al. [21], also deviates from unit-level exploration through loop outlining. However, as stated by the authors themselves, this solution introduces additional timing penalties due to extra function calls and parameter passing. In contrast, our approach directly orchestrates the pass manager to apply different optimization sequences per function within the same compilation unit, avoiding the need for outlining and its associated overheads. Moreover, it does not require the full back-end execution to estimate the WCET for each possible optimization sequence, saving a considerable amount of compilation time.

2) Optimality analysis: As discussed later in Section IV, the exploration space of the inner loop policies is too large to run optimal algorithms. However, even considering the inner loop policy as optimal, the iterative loop of Algorithm 1 is potentially non-optimal. Picking the optimal optimization for each function does not necessarily imply optimality for the whole compilation unit. The reason lies in how the function call stack is optimized (e.g., how the parameters are passed), which may preclude additional optimizations. However, our experimental results of Section V show that the function-based approach outperforms other non-function-based approaches.

C. Refining LLVMTA's Loop Analysis

During the development of Lavinium, we incidentally discovered an issue in the LLVMTA WCET calculation in the presence of nested loops. In fact, LLVMTA incorrectly calculates the maximum iteration count for the inner loops, propagating, under certain conditions, the outer loop iteration count in the inner loops. We fixed this issue so that the version of LLVMTA used in Lavinium correctly identifies the loop bound of a nested loop, leading to more accurate and upper-bounded WCET estimations.

In addition to the incorrect propagation, we discovered that some loop bounds analyses caused crashes due to unhandled errors in the SCEV. Lavinium's pass manager instrumentation tackles this by exploiting some wrappers for caching purposes around analysis components to speed up the rescheduling. This solved the issues with the SCEV and, in general, improved the analysis capabilities of our LLVMTA port. The impact of both refinements can be seen in Section V-B.

IV. Lavinium Exploration Policies

Lavinium has implemented several strategies for examining optimization sequences. The following sections detail the current policies in place. Specifically, we will discuss several state-of-the-art strategies that have been re-implemented and some newly developed or modified methods inspired by existing research. We have incorporated two state-of-the-art policies (Random and Genetic). Additionally, we have re-implemented the Association and Association with Cleaning policies [9], making modifications to improve their effectiveness. Finally, we have introduced three policies unexplored in the state-of-the-art: Greedy, Cartesian, and Cartesian Pruned.

A. Stochastic policies

These policies are implementations of stochastic algorithms that explore the optimization space via random sampling and genetic sampling.

1) Random: Firstly introduced by Puaut et al. [21], the random policy is the most trivial stochastic method for deriving the best sequence of optimizations. The algorithm generates random sequences in which each pass is extracted randomly from the sequence of optimization passes available. After *n* sequences have been explored, the sequence with the lowest WCET is returned.

2) Genetic: Another stochastic approach, firstly implemented by Puaut et al. [21] is based on the method of genetic algorithms. The process works as follows:

1) The first step of the genetic algorithm creates an initial population of *n* individuals, i.e., sequences of

passes, using the random method proposed earlier (Section IV-A1).
2) The population is evaluated with the static analyzer, and only a selection of the population is used as parents for the next generation.
3) The selected population is then used for breeding a new generation. Breeding is a two-step process that applies *crossover* and *mutation*. With crossover, two parents are chosen among the selected population, their sequences are split and swapped to create a new sequence. The new sequence then is subject to mutation with a certain probability *p*. With mutation, one of the passes in the newborn sequence is randomly substituted with another pass from the set of available passes.
4) After breeding, we obtain another population *n* individuals. The new population is evaluated again, and the process restarts from step 2. The cycle continues until a predefined sample size *s* is reached.

B. Association-rule-based Policies

The main objective of the association-rule-based policies [9] (called *Association* for short) is to characterize each optimization pass P depending on its impact as either *positive* (P^+), *neutral* (P°), or *negative* (P^-). The intuition behind this solution is to use the pass characterization to apply a weighting scheme based on the theory of association rule learning [22].

After a population of pass sequences has been analyzed (typically done with a random approach, as described in Section IV-A1), the proposed characterization defines a partial ordering among the sequences by generating a lattice upon their sets of passes[1]. Notice how this model disregards pass ordering. Therefore, although the solution can effectively characterize passes, it will likely miss higher-order interactions, meaning that it does not correlate the ordering of the sequence, but just whether two passes in the same sequence have a positive/negative impact.

1) Passes characterization: When exploring the original population, the algorithm builds up the lattice L by characterizing one sequence of passes at a time. Each node in the lattice is a set of passes S_i corresponding to a sequence s_i (notice that it could be that for multiple sequences s_i, s_j such that $s_i \neq s_j$ associated to the same set S_i) and WCET_i, where each pass has a characterization $C \in \{+, -, \circ\}$.

At the beginning, the lattice is only composed of one node, corresponding to the empty set $\varnothing$, with the baseline $\text{WCET}_\varnothing$ estimated on the original unoptimized code. Then, for each new sequence s_i, the algorithm assigns a characterization C to each pass in the new set S_i. We define the set S_i^{sub} of the largest subsets S_j of S_i in the lattice as:

$$S_i^{\text{sub}} = \{S_j \text{ s.t. } \forall S_k \in L, |S_j \cap S_i| \geq |S_k \cap S_i|\} \quad (1)$$

We say that S_i is characterized as:

- *positive* ($+$) if $\text{WCET}_i < \text{WCET}_j, \forall S_j \in S_i^{\text{sub}}$.
- *neutral* ($\circ$) if $\text{WCET}_i \leq \text{WCET}_j, \forall S_j \in S_i^{\text{sub}}$ and $\exists S_k \in S_i^{\text{sub}}$ s.t. $\text{WCET}_i = \text{WCET}_k$.
- *negative* ($-$) if $\exists S_j \in S_i^{\text{sub}}$ s.t. $\text{WCET}_i > \text{WCET}_j$.

If $|S_i^{\text{sub}}| = 1$, then we have a single greatest subset S_j of S_i (possibly the empty set $\varnothing$). In this case, each common pass $p_i \in S_i \cap S_j$ inherits its characterization from S_j. Non-common passes, instead, are characterized depending on the characterization of S_i. For instance, if we have the new set $S_i = \{A, B\}$ and its largest subset is $S_j = \{A^\circ\}$, and $\text{WCET}_i > \text{WCET}_j$, then we characterize $S_i = \{A^\circ, B^+\}$.

Else, if $|S_i^{\text{sub}}| = n > 1$, we write $S_i^{\text{sub}} = \{S_1, ..., S_n\}$ and we characterize each pass of S_i as follows:

- All passes $p \in S_i \setminus \bigcup_{j=1}^{n} S_j$ are characterized according to the characterization of the set S_i.
- All passes p for which $|\{S_j \in S_i^{\text{sub}} \text{ s.t. } p \in S_j\}| = 1$, i.e., belonging to only one subset, retain the characterization of their set, similarly to the case with a single largest subset.
- All the other passes, i.e. all passes p for which $|\{S_j \in S_i^{\text{sub}} \text{ s.t. } p \in S_j\}| > 1$, retain the most frequent characterization among the sets $S_j \ni p$. Ties are solved by looking at the characterization of the set S_i.

2) Passes Weighting: After the lattice is created on the initial population, the algorithm described in the original work [9] assigns to each pass P_i a weight w_i, computed as follows:

$$w_i = \frac{N_i^+}{max(1, N_i^\circ)} \times 0.85 + 0.15 \quad (2)$$

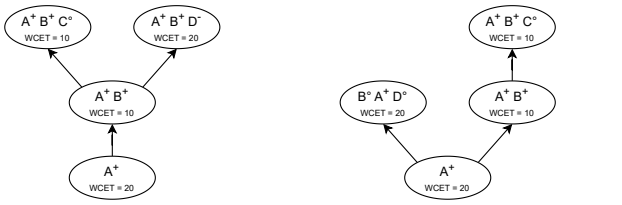
where N_i^+ is the number of times in which the pass P_i has been characterized as positive and N_i° is the number of times in which P_i has been characterized as neutral.

Once the weights are set, new sequences of passes are created with a weighted random selection. Then, after a new sequence is analyzed, the algorithm adds it to the lattice to better characterize its passes. Finally, weights w_i are updated for each pass in the new sequence after each analysis. The algorithm iterates until a user-defined number of sequences is reached. Notice how this scheme avoids passes that have been frequently characterized as neutral, while fostering the exploration of passes that have been frequently observed as positive or that do not appear frequently in the lattice.

3) Sequence Cleaning: Since the objective of the algorithm is to characterize as best as possible the passes that have a neutral impact on the compilation. The literature proposed a so-called *cleaning* phase [23] to do so. After a sequence of optimizations has been analyzed, the cleaning phase removes one pass at each step from the original sequence. If the WCET obtained from the resulting sequence is greater than or equal to the WCET of the original one, the pass is removed from the next step of the cleaning. Otherwise, it simply remains in the sequence. Each sequence is then analyzed again so that the characterization detects neutral passes and increases the confidence of the characterization of positive/negative passes.

4) Extending the Association Rule to Sequences: One of the main drawbacks of the approach presented so far is that it

[1]The sets are built with the unique passes of the original sequence



(a) Lattice constructed by the standard set-based characterization of the Association policy.

(b) Lattice constructed by the new sequence-based characterization of the Association policy.

Fig. 2. Difference between the lattices constructed by the original set-based characterization and by the new sequence-based characterization that accounts for optimization ordering.

"demotes" pass sequences into sets to build the lattice. As a consequence, any influence of the ordering of passes is simply disregarded. The problem of pass ordering (also called *phase ordering*) is one of the open problems of compiler autotuning, and many works showed the efficacy of running phase ordering when performing optimization selection, as surveyed in [2].

To tackle this shortcoming, our work extends the association rule algorithm by implementing the characterization lattice considering pass ordering. The underlying idea remains the same but, instead of subsets, we use subsequences to determine the partial ordering in the lattice. This new solution creates a denser lattice. In fact, in the original algorithm, two sequences with the same passes – but in different orders – would collapse into the same set, while in our case, we would have different nodes in the lattice.

As an example, consider the following pairs representing the sequences and their associated WCETs: $\langle A, 20\rangle$, $\langle AB, 10\rangle$, $\langle ABC, 10\rangle$, and $\langle BAD, 20\rangle$. With the original approach, we would obtain the lattice in Figure 2a, while, with our extension that accounts for pass ordering, we obtain the lattice in Figure 2b that provides more insight into the interaction among the passes. In particular, with our improved methodology, we can observe that there is a dependency between pass A and pass B so that applying A before B yields a lower WCET than B before A, and C and D are neutral passes (e.g. `ir-normalizer`). This kind of interaction is simply lost with the original approach, which would instead characterize D as a negative pass.

Moreover, the original algorithm does not state how to deal with two sequences with the same passes in a different order (i.e., same sequence and different sets). Following from the previous example, if we also had the characterization of $\langle BA, 20\rangle$ it would collapse in the set $\{A, B\}$. In this case, the original specification does not state which of the two values to keep. For this reason and for the drawbacks stated in Section IV-B4, we implemented only the extended version of Association with sequences in Lavinium.

C. Deterministic Policies

The policies presented in Section IV-A and Section IV-B are non-deterministic due to their reliance on stochastic factors for exploring the optimization space. As an alternative, we include three deterministic policies in Lavinium.

1) Greedy: The Greedy policy is one of the simplest methods for exploring the optimization space deterministically with reasonable overhead. Our Greedy algorithm takes as input the available passes and a target exploration depth. From there, it builds a sequence of passes starting from an empty sequence. The exploration is performed by finding the pass that optimizes the WCET at each depth, halting when the target depth is reached. Given a target depth d and the number of available passes n, Greedy performs a linear amount ($d \times n$) of WCET estimations. Yet, this algorithm has the huge disadvantage of missing inter-pass dependencies. It might be the case, in fact, that to run a pass B we need to run a preprocessing pass A, which could also worsen the WCET. Therefore, a third pass C may be chosen instead of A and B. However, it is possible that $WCET_{AB} < WCET_C$, hence the exploration performed by Greedy will most likely get stuck in a local optimum.

2) Cartesian: Another deterministic method to explore the optimization space is the Cartesian algorithm. Without diving into the details, this solution consists of a full exploration of the optimization space, finding the WCET of all the possible sequences of passes until a certain depth is reached. This algorithm is guaranteed to find the sequence of passes that minimizes the WCET. However, the computational cost is too high to make the implementation of this strategy actually feasible in a real-world scenario.

3) Cartesian with Pruning: Since the complete exploration of the optimization space is unfeasible, we implement a revised version of the Cartesian algorithm that is made of two phases: a cartesian pruned dependency exploration and a greedy optimizer. The main idea behind our new Cartesian pruned algorithm is that we want to perform what we call a preliminary cartesian pruned fixed-point dependency exploration of the optimization space to find dependencies among passes that would not be captured by the greedy algorithm. Once such dependencies are found, they are grouped into subsequences of passes that the greedy algorithm uses as building blocks for the final sequence.

The cartesian dependency exploration is performed by pruning the exploration space, removing *commutative* sequences of passes, and pass dependency is expressed by means of non-commutative sequences.

Definition 1 (Commutative Sequences)**.** *Two sequences s_i and s_j are **commutative** if:*

$$WCET_{s_i,s_j} = WCET_{s_j,s_i} \tag{3}$$

where $WCET_{x,y}$ is the WCET obtained applying, in order, sequence x and then sequence y.

Definition 2 (Dependent Sequences)**.** *Given two non-commutative sequences s_i and s_j, if:*

- *$WCET_{s_i} > WCET_{s_i,s_j}$, we say that s_j has a **positive dependency** on s_i.*
- *$WCET_{s_i} < WCET_{s_i,s_j}$, we say that s_j has a **negative dependency** on s_i.*

Sequences with positive dependencies are trivially kept, as we wish to obtain sequences that improve the WCET. On the other hand, we also keep negative dependencies since there can be cases in which several passes are needed as preprocessing passes in order to achieve better performance. For instance, consider a transformation A that requires a preprocessing sequence of passes s. We experimentally found that it is possible that $\text{WCET}_s > \text{WCET}_{baseline} > \text{WCET}_A > \text{WCET}_{sA}$. In our instance, we had that `gvn` and `licm` increase the WCET. However, when we schedule them before `loop-unroll`, we achieve a lower WCET. This observation implies that the WCET is not decreasing monotone and therefore we may also need to "pay" for the exploration of sequences of passes that worsen the WCET to achieve a better result.

Given the set of available passes P, the dependency exploration procedure begins by finding positive or negative dependencies among sequences composed of only a single pass, i.e., $s_i = \langle p_i \rangle$ and $s_j = \langle p_j \rangle$, with $p_i, p_j \in P$. Once the set of dependent sequences S is created, keep running the dependency analysis, finding all the dependent sequences $s_i p_i$ for all $s_i \in S$ and $p_i \in P$. All the newly found dependent sequences are added to S. The dependency exploration terminates upon reaching a fixed point, i.e., when no sequences are added to S when concatenated with all the passes $p_i \in P$. In fact, if appending each pass at the end of each sequence does not improve the WCET, it means that all those sequences will be pruned. Once the dependency exploration phase ends, the sequences found are added to the exploration space of the greedy algorithm. Hence, at each step, the algorithm finds the pass or sequence that provides the greatest improvement in the WCET, similarly to what we described in Section IV-C1.

V. Experimental Campaign

In order to check the quality of the WCET estimation and, in general, the benefits of Lavinium, we run several analyses and real-board experiments.

A. System Model

Table I summarizes, for each system model used for the experiments, the WCET analysis methodology, core pipeline, cache, and ISA, and – in the case of LLVMTA – specifies additional features, including the abstract model with an abstract address space representation, estimated before linking (pre-link). In particular, we ran the following experiments:

- Section V-B validates Lavinium against the original LLVMTA (using the RV-abs-val setup) to check whether our compilation flow still provides WCET estimations in line with LLVMTA.
- Section V-C measures the WCET improvement of the proposed exploration of compiler optimization sequences using the RV-abs-exp setup.

TABLE I
System Models used in the different experimental campaigns.

Identifier	Configuration	
RV-abs-val	WCET Tool:	LLVMTA
	Pipline:	in-order 5-stage pipeline
	Cache:	separated i-cache and d-cache, LRU, line size 16, associativity 2
	Architecture:	RISC-V 32-bit
	Limitations:	Abstract Model and Addresses, Pre-link
RV-abs-exp	WCET Tool:	LLVMTA
	Pipline:	in-order 5-stage pipeline
	Cache:	cacheless memory architecture
	Architecture:	RISC-V 32-bit
	Limitations:	Abstract Model and Addresses, Pre-link
SP-ait [9]	WCET Tool:	AbsInt aiT
	Pipline:	in-order 7-stage pipeline
	Cache:	16 KB 2-way instruction and data caches
	Architecture:	LEON3 32-bit SPARC V8
RV-mcu	WCET Tool:	Real hardware measure.
	Pipline:	in-order 2-stage pipeline
	Cache:	cacheless memory architecture
	Architecture:	RISC-V 32-bit

- Section V-D benchmarks the time required for running Lavinium, showcasing the speedup of the compilation and exploration compared to traditional strategies.
- Section V-F illustrates the observed execution time reduction by running on a real board (setup RV-mcu) the binaries produced by Lavinium.

B. Validation of Lavinium against LLVMTA

The goal of the first experimental evaluation is to verify whether the port of the LLVMTA analyzer into the Lavinium compilation flow was successful.

We compiled the set of `testcases` provided by the LLVMTA framework separately with the original LLVMTA toolchain and Lavinium. In particular, we used the same RISC-V configuration of the official LLVMTA repository for testing both frameworks, i.e., with `inorder` microarchitecture and `separatecaches` memory type (RV-abs-val). Although the specific architecture is not particularly relevant for this evaluation, we chose the RISC-V 32-bit architecture to ensure consistency with subsequent experiments.

As expected, Lavinium manifested an improvement in the successful generation of a WCET estimation. In fact, LLVMTA failed to provide a WCET estimation for 8 benchmarks – mostly due to SCEV errors, as described in Section III-C – and provided an unbounded result for 10 benchmarks out of a total of 34 benchmarks. Conversely, Lavinium failed to provide a WCET estimation for only 1 benchmark and provided an unbounded result for 1 benchmark. In some cases, Lavinium provides a more pessimistic WCET estimate due to: 1) the porting to a new LLVM version 17, which may have had changed internal functions and data structures used by LLVMTA; 2) the preprocessing passes required by LLVMTA, which could be slightly different; 3) the bug in LLVMTA previously described in Section III-C that causes `LoopBoundsInfo` analysis pass to generate potentially un-

derestimated WCET. The data confirms that Lavinium provides estimations at least as conservative as LLVMTA.

C. Lavinium Exploration Results

To quantify the benefits introduced by Lavinium, we evaluate the policies against different baselines using the RV-abs-exp setup, then the outcome is compared with the results reported by Dardaillon et al. [9] (setup SP-ait), which is the most recent article having the same focus. We run the well-known Mälardalen benchmark suite [24] widely used by various WCET analysis tools and the PolyBench [25]. Each benchmark is provided as a single C source file with, eventually, some header files. To establish the baselines, each benchmark was initially compiled using the four default optimization levels O0, O1, O2, and O3, and the WCET was estimated with a standard LLVMTA. Then, Lavinium is run with the strategies previously described in Section IV with the exception of Cartesian. Cartesian was excluded because, as expected, it is not computationally feasible when the exploration depth exceeds 4, and the results for smaller depths were insignificant. The Greedy strategy was scheduled to run until it reached a depth equal to the number of available passes (36 unique O3 passes in our case).

We configured all parameters of the exploration policies to be identical to the state-of-the-art paper [9]: genetic mutation rate at 15%, same weight scheme for both association and cleaning. All non-deterministic strategies (*association*, *clean*, *random*, and *genetic*) had the budget to generate $1\,000$ sequences of passes for each analyzed function (i.e., $1\,000\times n$, where n is the number of functions in the benchmark), with a sequence length of 10 passes, and the randomness source was seeded with unique values at each Lavinium inner loop iteration. The only difference with previous studies [9] is in the maximum sequence length: while the state-of-the-art paper used 50, we selected 10 for our test case. This decision was intentional, as shorter sequences allowed us to analyze them more effectively, as explained in Section V-E. This choice did not provide any advantage to our tool compared to previous works, as fewer passes were tested for each sequence. Moreover, Lavinium policies did not have any early stopping conditions, meaning that each exploration was run until completion. For this experiment, the RV-abs-exp configuration was chosen to reflect the microarchitectural features of RV-mcu, i.e., the evaluation MCU we used for the experiments described in Section V-F.

Table II presents the results of the benchmarks reporting the WCET of each configuration[2]. At least one of the Lavinium policies scored better than the baseline in 32 cases out of 36 benchmarks. In one case, `bs`, the result was identical to the baseline, due to the simplicity of the benchmark. In one other case, `fac`, LLVMTA cannot calculate the WCET[3]. The only non-trivial benchmark where Lavinium and O3 have scored the same results are `sqrt` and `jacobi-1d`.

We aggregated the relative results with the same strategy as the previous work [9], although we recognize that a fair comparison is difficult due to differences in targeted hardware, estimation tool, and benchmark selection – which overlaps but does not exactly match that of Dardaillon et al. [9]. Thus, aggregated results cannot be easily compared with their works, which represents a limitation of this analysis. Currently, we report these preliminary results for reference and to maintain consistency with earlier studies.

Table III reports the harmonic mean of the WCETs of each strategy, normalized with $\frac{\text{WCET}_{\text{Lavinium}}}{\text{WCET}_{Ox}} \times 100$, where $x \in \{0, 3\}$. All strategies considerably improve the WCET compared to O0 and O3. Association and Clean are the best policies, yet Pruned still finds better solutions in some benchmarks (e.g., `crc`). It is important to note that the harmonic mean metric can be significantly biased depending on the benchmark pool used, and it was selected to enable the comparison with the state of the art. For example, in this context, the benchmark *bs* contributes 100% to the mean, while *select* contributes only 49%. The results of *random* may be affected by a non-negligible statistical error due to the reduced sample space relative to the size of the exploration space, yet we do not expect significant differences from previous works.

Additionally, we have observed that the PolyBench suite is generally harder to optimize compared to the simpler Malardalen benchmarks. When examining the ratios of standard optimizations from O0 to O3, we find that *bs* speeds up by $30\times$, *matmult* by $10\times$, whereas all benchmarks in the PolyBench speed up by less than $2\times$.

D. Lavinium overhead analysis

The exploration process, as expected, introduces a significant overhead to the compilation. On a standard laptop, the compilation time for each benchmark and for each configuration provided in Table II spans from 5–10 minutes to several hours, depending on the benchmark size. While this is orders of magnitude longer than a typical compilation, we considered it acceptable since WCET optimization is not intended for everyday development, but rather as a final step in the implementation of a real-time system. We then compared the time required to compile the benchmarks with Lavinium and with the traditional approaches. Indeed, Lavinium schedules optimization internally in the back-end, while traditional approaches consider the compiler as a black box and tune its command line parameters. For this experiment, we measured the time required to compile each benchmark 100 times using the O3 optimization level. We then ran Lavinium with an ad-hoc strategy that suggests the same O3 passes at each of the 100 iterations: in this way, the time required to compile and optimize the benchmarks between the two strategies can be fairly compared. Table IV reports the average time and the standard deviation for the black box and Lavinium approaches. The Lavinium approach is considerably faster, achieving a speedup of $2.41\times$.

[2]The dataset with the raw results of the experiments can be found at: `https://doi.org/10.5281/zenodo.17227500`

[3]The benchmark contains volatile variables which cannot be analyzed by LLVMTA.

TABLE II

THIS TABLE REPORTS THE WCET CALCULATED BY LLVMTA WITH RV-ABS-EXP CONFIGURATION FOR EACH BENCHMARK, USING STANDARD OPTIMIZATION LEVELS FROM O0 TO O3 IN THE BASELINE COLUMN. THE LAVINIUM COLUMN SHOWS THE WCET FOR EACH BENCHMARK CALCULATED BY LLVMTA WITH RV-ABS-EXP CONFIGURATION USING DIFFERENT STRATEGIES. THE BEST WCET FOR EACH BENCHMARK IS HIGHLIGHTED IN BOLD.

BM	Baseline		Lavinium		BM	Baseline		Lavinium		BM	Baseline		Lavinium	
	O0	3 313	Association	**133**		O0	17 005 207	association	9 353 099		O0	2 798 814	association	**2 344 758**
	O1	**133**	Clean	**133**		O1	17 475 867	clean	**9 351 212**		O1	2 530 878	clean	**2 344 758**
bs	O2	**133**	Genetic	**133**	2mm	O2	16 462 718	genetic	9 353 099	gesummv	O2	2 625 587	genetic	**2 344 758**
	O3	**133**	Greedy	**133**		O3	16 462 718	greedy	13 624 090		O3	2 625 587	greedy	2 346 118
			Pruned	**133**				pruned	9 469 961				pruned	**2 344 758**
			Random	**133**				random	9 364 642				random	**2 344 758**
	O0	7 222 807	Association	**6 006 727**		O0	24 574 666	association	**12 833 634**		O0	99 900 220	association	**72 768 085**
	O1	7 016 727	Clean	**6 006 727**		O1	25 326 356	clean	12 942 687		O1	112 899 133	clean	**72 768 085**
bsort100	O2	7 016 727	Genetic	6 220 867	3mm	O2	24 019 678	genetic	**12 833 634**	heat-3d	O2	114 146 533	genetic	72 773 825
	O3	7 016 727	Greedy	6 412 767		O3	24 019 678	greedy	18 714 752		O3	114 146 533	greedy	73 176 449
			Pruned	**6 006 727**				pruned	12 942 687				pruned	72 770 505
			Random	6 017 857				random	**12 833 634**				random	72 774 235
	O0	2 567 917	Association	**625 017**		O0	68 430 523	association	**49 487 749**		O0	1 206 298	association	**1 204 128**
	O1	672 237	Clean	625 037		O1	69 329 593	clean	49 494 757		O1	1 205 368	clean	**1 204 128**
crc	O2	672 237	Genetic	627 597	adi	O2	62 896 222	genetic	49 494 757	jacobi-1d	O2	**1 204 128**	genetic	**1 204 128**
	O3	672 237	Greedy	630 157		O3	62 896 222	greedy	56 759 985		O3	**1 204 128**	greedy	1 205 368
			Pruned	**625 017**				pruned	49 494 757				pruned	**1 204 128**
			Random	625 037				random	49 494 757				random	**1 204 128**
	O0	∞	Association	∞		O0	4 099 159	association	**3 619 002**		O0	52 788 013	association	51 006 437
	O1	∞	Clean	∞		O1	4 264 539	clean	**3 619 002**		O1	60 378 737	clean	51 006 437
fac	O2	∞	Genetic	∞	atax	O2	3 919 102	genetic	**3 619 002**	jacobi-2d	O2	60 378 737	genetic	**49 829 737**
	O3	∞	Greedy	∞		O3	3 919 102	greedy	3 631 682		O3	60 378 737	greedy	50 962 477
			Pruned	∞				pruned	3 622 122				pruned	51 006 437
			Random	∞				random	**3 619 002**				random	51 006 437
	O0	17566	Association	10 316		O0	3 874 780	association	**3 432 518**		O0	4 250 273	association	**3 702 649**
	O1	11 436	Clean	**3 096**		O1	3 989 940	clean	**3 432 518**		O1	3 932 249	clean	**3 702 649**
fibcall	O2	11 436	Genetic	10 946	bicg	O2	3 615 441	genetic	**3 432 518**	mvt	O2	3 995 559	genetic	**3 702 649**
	O3	11 436	Greedy	11 436		O3	3 615 441	greedy	3 442 818		O3	3 995 559	greedy	3 705 929
			Pruned	10 316				pruned	**3 432 518**				pruned	**3 702 649**
			Random	11 436				random	**3 432 518**				random	**3 702 649**
	O0	8 736 907	Association	**6 909 907**		O0	26 934 292	association	**15 101 487**		O0	199 354 873	association	**131 560 043**
	O1	7 091 937	Clean	**6 909 907**		O1	27 824 242	clean	15 102 147		O1	205 763 023	clean	**131 560 043**
fir	O2	7 091 937	Genetic	**6 909 907**	covariance	O2	26 255 320	genetic	15 303 378	nussinov	O2	163 446 713	genetic	132 803 503
	O3	7 091 937	Greedy	**6 909 907**		O3	26 255 320	greedy	24 847 070		O3	163 446 713	greedy	148 835 593
			Pruned	**6 909 907**				pruned	15 338 196				pruned	132 274 353
			Random	**6 909 907**				random	16 001 776				random	132 096 223
	O0	57 243	Association	**8 773**		O0	18 836 832	association	**9 116 149**		O0	79 207 489	association	**74 589 000**
	O1	42 763	Clean	**8 773**		O1	19 277 772	clean	**9 116 149**		O1	88 508 920	clean	**74 589 000**
insertsort	O2	52 543	Genetic	39 453	doitgen	O2	9 945 968	genetic	9 213 715	seidel-2d	O2	83 930 710	genetic	**74 589 000**
	O3	52 543	Greedy	39 453		O3	9 945 968	greedy	10 062 164		O3	83 930 710	greedy	75 415 371
			Pruned	39 453				pruned	9 258 137				pruned	**74 589 000**
			Random	39 453				random	9 168 717				random	**74 589 000**
	O0	120 557	Association	**14 537**		O0	3 237 533	association	**3 081 242**		O0	26 069 061	association	**22 236 824**
	O1	18 817	Clean	**14 537**		O1	3 121 523	clean	**3 081 242**		O1	24 317 533	clean	**22 236 824**
matmult	O2	18 817	Genetic	**14 537**	durbin	O2	3 123 853	genetic	3 110 482	symm	O2	24 304 924	genetic	**22 236 824**
	O3	18 817	Greedy	**14 537**		O3	3 123 853	greedy	3 120 293		O3	24 304 924	greedy	23 300 273
			Pruned	**14 537**				pruned	**3 081 242**				pruned	**22 236 824**
			Random	**14 537**				random	3 090 402				random	**22 236 824**
	O0	388 146	Association	**128 856**		O0	45 627 644	association	44 616 597		O0	42 063 406	association	**35 834 060**
	O1	199 926	Clean	**128 856**		O1	47 388 322	clean	44 616 597		O1	38 856 390	clean	**35 834 060**
ns	O2	199 926	Genetic	128916	fdtd-2d	O2	49 383 387	genetic	44 616 597	syr2k	O2	38 857 822	genetic	**35 834 060**
	O3	199 926	Greedy	129846		O3	49 383 387	greedy	46 164 286		O3	38 857 822	greedy	35 836 230
			Pruned	**128 856**				pruned	44 616 597				pruned	**35 834 060**
			Random	128916				random	**44 541 047**				random	**35 834 060**
	O0	55 689	Association	**50 518**		O0	194 310 473	association	**120 541 343**		O0	25 851 344	association	**21 080 887**
	O1	50 560	Clean	**50 518**		O1	187 236 363	clean	**120 541 343**		O1	23 090 777	clean	**21 080 887**
qurt	O2	50 560	Genetic	**50 518**	floyd-warshall	O2	136 820 453	genetic	**120 541 343**	syrk	O2	23 090 777	genetic	21 083 057
	O3	50 560	Greedy	**50 518**		O3	136 820 453	greedy	135 970 113		O3	23 090 777	greedy	21 083 057
			Pruned	**50 518**				pruned	**120 541 343**				pruned	**21 080 887**
			Random	**50 518**				random	**120 541 343**				random	**21 080 887**
	O0	675 397	Association	**671 537**		O0	20 083 599	association	**12 298 140**		O0	3 045 570	association	**2 570 221**
	O1	1 348 827	Clean	**671 537**		O1	20 631 159	clean	**12 298 140**		O1	2 738 401	clean	**2 570 221**
select	O2	1 348 827	Genetic	**671 537**	gemm	O2	18 929 754	genetic	**12 298 140**	trisolv	O2	2 742 993	genetic	**2 570 221**
	O3	1 348 827	Greedy	673 317		O3	18 929 754	greedy	17 263 674		O3	2 742 993	greedy	2 588 663
			Pruned	**671 537**				pruned	12 324 180				pruned	**2 570 221**
			Random	**671 537**				random	**12 298 140**				random	**2 570 221**
	O0	44 689	Association	**40 255**		O0	6 872 054	association	**6 291 658**		O0	20 142 111	association	**17 684 295**
	O1	43 055	Clean	**40 255**		O1	7 039 784	clean	**6 291 658**		O1	20 026 675	clean	**17 684 295**
sqrt	O2	**40 255**	Genetic	**40 255**	gemver	O2	6 660 803	genetic	**6 291 658**	trmm	O2	19 955 065	genetic	**17 684 295**
	O3	**40 255**	Greedy	**40 255**		O3	6 660 803	greedy	6 310 928		O3	19 955 065	greedy	17 725 266
			Pruned	**40 255**				pruned	**6 291 658**				pruned	**17 684 295**
			Random	**40 255**				random	**6 291 658**				random	17 684 925

TABLE III

THE HARMONIC MEAN OF THE WCET RATIO COMPARED TO THE STANDARD OPTIMIZATIONS O0 AND O3, COMPUTED ON RV-ABS-EXP. THE RESULTS ARE COMPARED WITH THE STATE OF THE ART (SOA), CONFIGURATION SP-AIT.

	Lavinium		SoA as reported by [9]
Benchmark	**O0**	**O3**	**O3**
Association	40.7%	73.3%	80.6%
Clean	38.9%	69.6%	79.7%
Genetic	43.3%	81.5%	82.4%
Greedy	45.2%	87.7%	-
Pruned	43.3%	81.4%	-
Random	43.4%	81.7%	83.2%

TABLE IV

TIME REQUIRED TO FIND THE SOLUTION BY THE BLACK BOX APPROACH AND BY LAVINIUM WHEN EXECUTED ON 100 ITERATIONS.

Configuration	**Black box**	**Lavinium**
Average Time	7.98 s	3.30 s
Standard Deviation	1.99 s	1.75 s

E. Pass sequence characterization

For each benchmark and exploration policy, we have gathered the sequences of optimizations that minimized the WCET function-wise. Let us call this multiset $\mathrm{SEQ}_{\min}$. These sequences were further analyzed to identify the most common subsequences. For each subsequence, we have calculated its

frequency as the ratio between the number of unique occurrences in each sequence of SEQ_{min} and the size of SEQ_{min}. The ten most frequent passes present in the optimization sequence can be grouped principally into two categories[4] The first group of passes consists of simplification passes that remove redundant operations or combine multiple instructions into simpler ones. This group contains `gvn` (16%), `sroa` (16%), `instcombine` (14%), and `mldst-motion` (12%). The second group is composed of passes that are loop-related and tend to remove or simplify the instruction inside the loop: `licm` (28%), `indvars` (16%), `loop-deletion` (15%), `loop-vectorize` (14%), `loop-unroll` (13%), and `loop-idiom` (12%). The pass that appears most frequently in the optimization sequence is `licm`. This is expected since `licm` optimizes loops by moving operations outside of them, which helps with other loop transformations. Upon analyzing the subsequences of length greater than one, we found that their frequency is less than 2%. This provides strong evidence that there are no subsequences of two or more passes that are consistently beneficial to the WCET. Therefore, it is necessary to employ an exploratory approach on a case-by-case basis.

F. Lavinium optimization sequences on real hardware

The final experiment has the objective of providing an estimate of the impact of the Lavinium exploration policies on the execution time observable on real hardware. To do so, we measured the execution time of Lavinium-optimized benchmarks in the RV-mcu setup to extract the Worst-Case Observed Time (WCOT). The RV-mcu target is the FPB-R9A02G021 RISC-V MCU Fast Prototyping Board by Renesas, equipped with a R9A02G021 microcontroller based on a single 48MHz RISC-V core with architecture RV32I[MACB].

We compiled a subset of the benchmarks used in Section V-C, slightly modified to allow random input generation to explore as many execution paths as possible. Each benchmark has been compiled by applying all Lavinium policies on the RV-abs-exp abstract machine, and emitting one executable for every applied policy targeting the aforementioned real-world MCU. More specifically, Lavinium was set up to emit LLVM IR files in which every function is optimized according to the selected policy. After all the benchmarks have been compiled, the `.ll` file is fed to the Renesas LLVM-based compilation toolchain, which generates the object file and links it to the testing software component. We ran each benchmark 10 000 times. Every run begins with a preparation phase exploiting a Pseudo-Random Number Generator (PRNG) to fill the input data of the benchmark. Then, the benchmark is executed, measuring its execution time thanks to the machine-mode cycle counter via the Control and Status Registers (CSRs) `mcycleh` and `mcycle`. These readings are designed to capture the clock cycles consumed by the benchmark while excluding the overhead of the experiment and timing instrumentation.

[4]The technical report, which can be found at `https://doi.org/10.5281/zenodo.17232204`, provides a more detailed description of the role of each pass in the compilation pipeline.

TABLE V
THIS TABLE REPORTS THE WCET OBSERVED RUNNING EACH BENCHMARK ON THE R9A02G021 MCU WITH RV-MCU CONFIGURATION. THE WCOT COLUMN SHOWS MAXIMUM OBSERVED EXECUTION TIME FOR EACH BENCHMARK, WHILE THE ACET COLUMN SHOWS THE AVERAGE-CASE OBSERVED EXECUTION TIME FOR EACH BENCHMARK. THE BEST WCOTS AND ACETS FOR EACH BENCHMARK ARE HIGHLIGHTED IN BOLD.

BM	Policy	WCOT	ACET	BM	Policy	WCOT	ACET
	None	336	326		None	2 189	1 393
	Association	**55**	**55**		Association	798	511
	Cleaning	**55**	**55**		Cleaning	**772**	**493**
bs	Genetic	**55**	**55**	insertsort	Genetic	804	509
	Greedy	57	57		Greedy	1 980	1 193
	Pruned	57	57		Pruned	2 071	1 291
	Random	**55**	**55**		Random	797	509
	None	237 251	227 569		None	6 000	5 998
	Association	227 647	218 086		Association	959	959
	Cleaning	207 351	198 130		Cleaning	959	959
bsort100	Genetic	227 647	218 086	matmult	Genetic	959	959
	Greedy	229 705	219 193		Greedy	971	971
	Pruned	**199 509**	**189 484**		Pruned	**957**	**957**
	Random	207 351	198 129		Random	959	959
	None	104 317	5 843		None	30 920	30 919
	Association	33 162	5 498		Association	9 783	9 783
	Cleaning	33 840	5 414		Cleaning	9 783	9 783
crc	Genetic	34 185	5 498	ns	Genetic	9 910	9 910
	Greedy	34 868	**5 406**		Greedy	30 765	30 764
	Pruned	33 159	5 510		Pruned	**9 782**	**9 782**
	Random	**33 083**	5 420		Random	9 785	9 785
	None	–	–		None	26 290	25 541
	Association	–	–		Association	26 291	25 652
	Cleaning	–	–		Cleaning	26 335	25 679
fac	Genetic	–	–	qurt	Genetic	26 338	25 682
	Greedy	–	–		Greedy	26 001	**25 254**
	Pruned	–	–		Pruned	**25 999**	25 283
	Random	–	–		Random	26 298	25 637
	None	837	463		None	**5 862**	**3 962**
	Association	**313**	**192**		Association	6 076	4 056
	Cleaning	761	427		Cleaning	6 086	4 056
fibcall	Genetic	733	413	select	Genetic	6 093	4 060
	Greedy	863	475		Greedy	5 972	3 985
	Pruned	733	411		Pruned	5 973	3 980
	Random	**313**	**192**		Random	6 080	4 071
	None	**912 687**	**447 762**		None	21 804	19 698
	Association	932 682	457 582		Association	21 495	19 359
	Cleaning	931 317	456 921		Cleaning	21 525	19 387
fir	Genetic	932 680	457 580	sqrt	Genetic	21 525	19 387
	Greedy	931 323	456 927		Greedy	**21 440**	**19 345**
	Pruned	932 676	457 576		Pruned	21 442	19 347
	Random	932 684	457 580		Random	21 495	19 365

The Average-Case Execution Time (ACET) and the Worst-Case Observed execution Time (WCOT) for each benchmark are reported in Table V. Indeed, LLVMTA's abstract model limits the interpretability of this comparison. Although the optimization sequences were produced targeting the RV-abs-exp abstract model, the results illustrate that the Lavinium static exploration reduced the execution time of multiple programs, also in the RV-mcu setup, since the optimized versions exhibited a sensible WCOT improvement. There are only two exceptions (`fir` and `select`) in which Lavinium WCOT exceeded 2–3% the default strategy. This result should not be considered negative because the true worst-case execution scenarios are difficult to reproduce experimentally. From the results, we also observe that achieving a better average-case observed execution time does not necessarily entail achieving a better worst-case observed execution time, as expected.

Overall, these experiments show that Lavinium explores optimization sequences that provide improved worst-case observed execution time also on real hardware, assuming that the abstract LLVMTA model reasonably mimics the target architecture, effectively enhancing the system's real-time properties.

VI. Limitations and Future Works

While Lavinium proves the effectiveness of WCET-driven compiler autotuning, we acknowledge some limitations, some of which can be addressed in future research.

The choice of LLVMTA as a WCET analyzer tool has both advantages and disadvantages. On the positive side, being LLVM-based, LLVMTA has been integrated within the Lavinium compilation loop flawlessly and shows promise in many aspects. For instance, since LLVMTA identifies the WCET critical path, this information can be exploited in the future to drive the exploration. Other LLVMTA-provided information can also be used in future works to extend Lavinium capabilities. On the other hand, the dependency of Lavinium on LLVMTA carries a set of limitations. First, Lavinium's results must be interpreted accounting for the abstract model used by LLVMTA. As discussed in Section V-F, wherein we show that Lavinium can be used to guide the optimization exploration, we must consider that the timing bounds produced by the tool at the current stage may not reflect the hardware WCET since abstract modeling may introduce biases with respect to real silicon. On this note, a second limitation is linked to LLVMTA MIR-level analysis, which only guesses addresses and memory layouts. Despite being proven effective, these analyses can only approximate actual hardware WCET estimations. A third LLVMTA limitation concerns the generalizability of its analysis. Although LLVMTA is open-source, it would be feasible, though not trivial, to extend it; it may not support complex hardware and software features (e.g., address translation for cache analysis, as well as considering the impact of context switches and interrupt management on the microarchitecture state).

To tackle LLVMTA limitations, Lavinium could be adapted to replace it with other WCET estimators. Future work could integrate Lavinium with more precise commercial analyzers (e.g., aiT) or hardware-in-the-loop estimation procedures [26]. To incorporate these tools, the inner loop of Lavinium should be modified to include the assembly emission phase. With this adjustment, Lavinium can directly work with the object file and link it to all other required artifacts before interacting with the external tool. The external tool can then be used similarly to LLVMTA, providing the Rescheduler with feedback on the quality of the selected pass. While this modification will enhance the accuracy of the WCET estimation, it will also slow down the compilation speed.

Our results show that function-level optimization exploration can further optimize the WCET with respect to standard global module-level explorations. A comparison between Table II results and new experiments using Lavinium applied per-module rather than per-function, in line with [9], would enable a more thorough evaluation, which we leave for future work. Additionally, unpredictable side effects may arise in the case of inter-function dependencies. For instance, once a function f_i is deemed optimal by a given policy, and a next function f_j is being optimized, some optimizations on f_j may increase the WCET of f_i and render the sequence used for compiling f_i suboptimal. Future studies involve a thorough investigation of this scenario.

There are also limitations intrinsic to the combinatorial nature of the optimization exploration problem. In fact, Lavinium is a heuristic-based toolchain and may not be able to determine the global optimum. For this reason, most policies in the literature use stochastic approaches, which may be unstable and produce diverse optimal sequences and WCET estimates. Finding deterministically the global optimum is indeed possible with Lavinium, which implements a complete optimization space exploration policy (Cartesian), but it is computationally infeasible for real-world applications. In this direction, Lavinium could benefit from the investigation of novel exploration policies and the use of parallel compilation, which is, however, not currently supported by LLVM. Additionally, at this stage, only LLVM IR-level passes have been analyzed. Backend-specific passes have not yet been explored and could be the object of future work.

VII. Conclusions

In this paper, we have introduced Lavinium, an LLVM-based tool that aims to optimize the WCET of a program by exploring the space of compiler optimization sequences. Conversely from previous works, Lavinium is completely integrated into LLVM and has several advantages: 1) a significantly lower compilation time, 2) the exploration of compiler optimizations per-function, rather than per-module, improving the granularity of the WCET optimization level, 3) all the advantages of the LLVM framework, integrating the LLVMTA static timing analyzer, and therefore enhancing the flexibility of the exploration. Lavinium implements and improves four state-of-the-art exploration algorithms, extending them to account for sequence ordering. Additionally, we implemented two deterministic exploration policies (greedy and cartesian) and the new *cartesian pruned*.

Our experiments tested several aspects of Lavinium. We validated Lavinium against the baseline LLVMTA configuration, showing that Lavinium can successfully estimate more WCETs (32) compared to the original LLVMTA (16). Then, we tested Lavinium on a suite of WCET benchmarks to compare the effectiveness of each policy and to verify the results provided by WCET-driven optimization sequences compared to standard optimization levels. The obtained WCET by Lavinium is up to 69% lower than O3. Furthermore, we compared the compilation time required by Lavinium and by standard approaches, assuming the compiler as a black box, obtaining a speedup of $2.41\times$. Finally, we ran the benchmarks on a real RISC-V board to verify that the sequences derived from Lavinium's abstract WCET optimization also improve the worst-case observed execution times.

Acknowledgments

This work has received funding by the ISOLDE Chips JU project (grant nr. 101112274).

References

[1] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), September 2018.

[3] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving wcet by optimizing worst-case paths. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 138–147, 2005.

[4] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving wcet by applying worst-case path optimizations. *Real-Time Systems*, 34(2):129–152, Oct 2006.

[5] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a wcet-aware c compiler. In *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 121–126, 2006.

[6] Mohamed Amine Khelassi and Yasmina Abdeddaïm. Impact of compilation optimization levels on execution time variability. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2024.

[7] Mohamed Abdel Maksoud and Jan Reineke. A compiler optimization to increase the efficiency of wcet analysis. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 87–96, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Martin Becker and Samarjit Chakraborty. Optimizing worst-case execution times using mainstream compilers. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, page 10–13, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Mickaël Dardaillon, Stefanos Skalistis, Isabelle Puaut, and Steven Derrien. Reconciling compiler optimizations and wcet estimation using iterative compilation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 133–145, 2019.

[10] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 97–106, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Sebastian Hahn, Michael Jacobs, Nils Hölscher, Kuan-Hsun Chen, Jian-Jia Chen, and Jan Reineke. LLVMTA: An LLVM-Based WCET Analysis Tool. In Clément Ballabriga, editor, *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)*, volume 103 of *Open Access Series in Informatics (OASIcs)*, pages 2:1–2:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[13] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 12–21, 1999.

[14] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[15] AbsInt. ait: The industry standard for static timing analysis. https://www.absint.com/ait/, 2002. [Accessed 05-03-2025].

[16] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[17] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007. Special issue on Experimental Software and Toolkits.

[18] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *Open Access Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[19] Stephan Thesing. Safe and precise wcet determination by abstract interpretation of pipeline models, 2004.

[20] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Rev.*, 12(1):28–36, March 2015.

[21] Isabelle Puaut, Mickaël Dardaillon, Christoph Cullmann, Gernot Gebhard, and Steven Derrien. Fine-Grain Iterative Compilation for WCET Estimation. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *Open Access Series in Informatics (OASIcs)*, pages 9:1–9:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[23] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.*, 9(3), October 2012.

[24] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[25] Scott Grauer-Gray et al. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, 2012.

[26] Marcus Lindner, Jorge Aparicio, Henrik Tjader, Per Lindgren, and Johan Eriksson. Hardware-in-the-loop based wcet analysis with klee. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 345–352, 2018.