2026 Design, Automation & Test in Europe Conference (DATE 2026)

VLIM: Verified Loop Interchange for Optimised Matrix Multiplication

Oliver Turner
Dept. of Computer Science, Durham University
DH13LE United Kingdom
oliver.turner2@durham.ac.uk

Shounak Chakraborty
Dept. of Computer Science, Durham University
DH13LE United Kingdom
shounak.chakraborty@durham.ac.uk

***Abstract*—Loop optimisations are essential for achieving high performance in modern computing, particularly for memory-intensive operations. However, while unverified optimisers achieve impressive speedups, their manual application is error-prone and challenging to verify, making them risky in high-assurance computing platforms. This paper introduces *VLIM*, a novel rewrite algebra, to overcome these difficulties, enabling the development and automatic verification of loop transformations within the Capla programming language, a formally defined front-end for the Compcert verified compiler. Our framework allows compiler developers to define rewrite rules, with correctness proofs automatically derived through rewrite composition, ensuring semantic preservation during optimisation. We demonstrate the effectiveness of our approach, *VLIM*, by implementing a loop interchange optimisation and evaluating its impact on matrix multiplication performance. Empirical analyses show significant performance improvements: for a 1000×1000 matrix, loop interchange using *VLIM* reduced runtime by 36.6% and 74.6% when compiled with Compcert and Clang, respectively. This work advances the state-of-the-art in verified compilation, offering a promising direction for developing high-performance, formally verified software.**

***Index Terms*—Loop Interchange, Loop Optimisation, Compiler Optimisation, Cache Misses, Formal Verification**

I. Introduction

The increasing disparity between processor and memory speeds, known as the "von Neumann bottleneck" or "memory wall" [1], [2], significantly challenges software developers. Exploiting modern hardware necessitates an in-depth understanding of processor cache architectures (considering critical design parameters: capacity, line size, associativity) [3], [4] to optimise memory access patterns and minimise latency in frequently executed code. However, this optimisation inherently introduces substantial low-level code complexity. Real-world optimisations like loop tiling or data prefetching demand extensive, error-prone profiling and fine-tuning due to heuristic reliance and hardware variations. Such low-level manipulation frequently obscures original program semantics, injecting subtle, hard-to-detect errors. Modern processor architectures, with intricate multi-level cache hierarchies, pipelined execution, and sophisticated speculative units, exacerbate manual optimisation challenges, risking performance regressions or critical behavioural errors.

Formal verification offers a powerful, rigorous approach to ensuring program correctness by constructing mathematical proofs that a program adheres to its intended behaviour for all inputs and execution scenarios [5]. Unlike empirical methods like extensive testing (finite subset of executions, subtle corner cases missed) [6], or static analysis (false positives, overlooked complex errors) [7], formal verification provides a much higher degree of confidence [8]. By embedding mathematical proofs directly in development, formal methods eliminate entire classes of hard-to-detect errors. This is crucial in safety-critical code, where minor specification deviations or subtle optimisation bugs can lead to significant consequences, from degradation to catastrophic failures. Despite the clear benefits of formal methods, a critical gap persists: current high-performance compilers and manual optimisation techniques, while achieving impressive speedups, often lack formal guarantees of correctness, making their deployment risky in high-assurance environments.

This paper presents *VLIM*, a novel framework facilitating correct-by-construction loop optimisations. *VLIM* directly addresses the aforementioned gap by introducing a sophisticated rewrite algebra enabling the systematic definition, application, and formal verification of a broad class of loop transformations within Capla, a formally defined front-end for the Compcert verified compiler [9], [10]. Unlike previous work on verified compilers that focuses on internal, fixed optimisation passes, our framework empowers library developers to define and compose precise rewrite rules for new optimisations, whose correctness proofs are automatically derived from the algebra's properties, ensuring semantic preservation in complex transformations. This approach bridges the gap between high-level, readable code and low-level, highly optimised implementations by providing a mechanised tool, formalised in the Rocq theorem prover [11], to guarantee that the optimised code behaves identically to its original, unoptimised counterpart.

We demonstrate the efficacy and generality of this approach by implementing a loop interchange optimisation to enhance data locality and cache utilisation, specifically evaluating its significant impact on matrix multiplication performance. While state-of-the-art unverified optimisers can achieve high performance, *VLIM*'s core novelty lies in delivering these critical speedups with unprecedented formal guarantees of correctness. *VLIM* significantly advances verified compilation by providing a robust, extensible pathway to high-performance, formally verified software.

978-3-9826741-1-7/DATE26

The main contributions of this paper are:

- A novel rewrite algebra for expressing and verifying loop transformations.
- A mechanised formalisation of the rewrite algebra in the Rocq theorem prover.
- A formally-verified loop interchange optimisation, showcased through its application to matrix multiplication.
- An empirical evaluation of the performance and cache improvements achieved by the verified transformation.

Overall, *VLIM introduces a novel rewrite algebra for verified loop optimisations in the Capla programming language, enabling automatic, semantically-preserving loop transformations.* Its efficacy is demonstrated by significantly improving matrix multiplication performance, notably reducing 1000×1000 matrix runtime by 36.6% (from 3.58s to 2.27s) for Compcert and by 74.6% (from 1.65s to 0.42s) for Clang.

II. Background and Motivation

Efficient memory access is critical, as memory operations significantly lag processor speeds, incurring hundreds of cycles of latency compared to arithmetic operations [12]. This performance gap is exacerbated by hierarchical memory systems. Data locality is paramount; good temporal and spatial locality allow data retrieval from faster cache levels, minimising costly cache misses and improving performance. For instance, as illustrated in Figure 1, the iteration order in matrix operations directly impacts cache stride and access patterns. Accessing elements row-major improves spatial locality by effectively utilising cache lines, compared to column-major access for row-stored matrices, thereby reducing cache misses and improving performance.

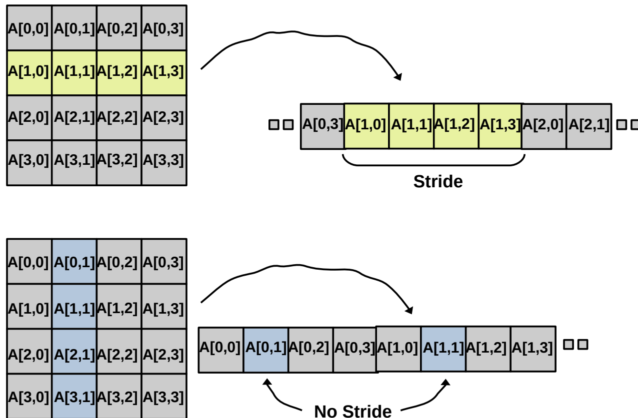

Fig. 1: Iteration order in a matrix affects cache stride and access pattern

Our work builds upon the foundation of the Compcert verified C compiler [13] and the Capla programming language [9]. Compcert is a pioneering formally verified optimising compiler, processes code through various stages, transforming Abstract Syntax Trees (ASTs) into intermediate representations (IRs) and finally to machine code. Critically, all these transformations are formally proven correct, ensuring full semantic preservation during compilation. Capla serves as an alternative front-end designed with formal semantics that explicitly compose with Compcert's correctness proofs, providing a high-level language for developing verified low-level computer algebra routines. This integration enables verified loop optimisations to be applied at a higher level of abstraction, leveraging Compcert's established trustworthiness.

Despite advances in compiler optimisation and formal verification, a critical gap remains: formally verifying user-applied, cache-aware loop transformations. Existing performance-focused studies, including those on loop tiling and cache optimisation, often lack formal correctness guarantees, hindering their deployment in high-assurance environments. Techniques like polyhedral compilation offer powerful, aggressive performance optimisation, but typically operate without formal proofs of their implementations. Our work, *VLIM*, addresses this by *synthesising verified compilation techniques to extend formal guarantees to powerful, user-applied rewrite optimisations previously lacking comprehensive verification*. This novel approach enables performance-critical loop transformations with high confidence in formally verified environments.

III. *VLIM*: Proposed Approach

This section details *VLIM*'s novel rewrite algebra for Capla, enabling verified, user-applied loop optimisations with automatically derived proofs.

```
Theorem function_spec args e' vres:
  eval_funcall ge (Internal f) args e' vres ->
  P args e' vres.
```

Fig. 2: The type of a proof of correctness of a Capla function.

```
Theorem function_spec R args e' vres:
  rewrite_is_correct R ->
  eval_funcall ge (Internal (R f)) args e' vres ->
  P args e' vres.
```

Fig. 3: The type of the fact we would like to recover from our rewrites.

A. Capla Architecture and Semantic Model

Capla is designed as a front-end language for the Compcert verified C compiler [9], [10], particularly suited for developing verified low-level computer algebra routines. While Compcert specifies its IRs using a small-step semantics, for reasoning about program properties in Capla, we utilise a more proof-friendly big-step semantics. This semantic model allows us to reason about the execution of statements and functions through a mutually recursive indexed inductive type. A proof about the behaviour of a Capla function, f, is typically formulated as a theorem, exemplified by the structure in Figure 2. This theorem asserts that for given input arguments, environment, and return value, the function's successful evaluation ('eval_funcall') implies a specific property P about these elements. Our objective is to define transformations R such that if applied successfully to f to yield f', the equivalent theorem holds for the rewritten function f', thereby formally preserving the original function's specification. This desired property of rewrites is conceptually illustrated in Figure 3, where '$rewrite_is_correct$' ensures that the transformed function maintains the desired semantic property.

```
Definition fn_rewrite_correct r :=
  forall f f',
  r f = OK f' ->
  forall g args e o,
  eval_funcall g (Internal f) args e o ->
  exists e',
    eval_funcall g (Internal f') args e' o /\
    restrict_mut (fn_penv f) e =
      restrict_mut (fn_penv f') e'.
```

Fig. 4: What it means to be a correct rewrite.

```
Record function_rewrite : Type :=
    mk_function_rewrite {
  rewrite: function -> res function;
  correct: fn_rewrite_correct guard rewrite;
}.
```

Fig. 5: The type of a complete rewrite bundle.

B. Rewrite Correctness

Central to *VLIM* is the formal definition of a "correct rewrite". Rewrites are defined as fallible functions of type ‘(function $- >$ res function)’, allowing them to express cases of non-applicability during optimisation. The fundamental essence of a correct rewrite is precisely captured by the ‘$fn_rewrite_correct$’ definition, as detailed in Figure 4. This definition formally states that if a rewrite ‘r’ successfully transforms an original function ‘f’ into a rewritten function ‘f’, then for any execution of ‘f’ producing output ‘o’ and environment ‘e’, there must exist an environment ‘e’ such that ‘f’ also evaluates to the same output ‘o’. Crucially, the mutable arguments of the original and rewritten functions must be identical in their respective output environments, ensuring semantic equivalence of side effects on observable memory. This specific requirement for mutable arguments aligns with Capla's copy-restore calling convention, where only mutable parameters are copied back to the caller. The use of an existential quantifier for the resulting environment ‘e’ is justified because the ‘$eval_funcall$’ operation (function evaluation) is deterministic, ensuring a successful rewrite does not introduce new error conditions or non-determinism. These correctness properties, along with the rewrite function itself, are formally bundled into a standard Rocq record called ‘$function_rewrite$’, which is presented in Figure 5.

```
Definition stmt_rewrite_id' s : res stmt := OK s.

Lemma stmt_rewrite_id_correct :
  stmt_rewrite_correct stmt_rewrite_id'.
Proof. by move=>> ? > [<-]. Qed.

Definition stmt_rewrite_id := {|
  rewrite := stmt_rewrite_id';
  correct := stmt_rewrite_id_correct;
|}.
```

Fig. 6: The trivial identity rewrite.

C. Rewrite Algebra

The design of *VLIM* incorporates a rewrite algebra, a formal system that significantly simplifies the development and composition of verified optimisations, leading to automated proof verification and re-composition. By defining foundational elements such as the identity rewrite (‘$stmt_rewrite_id$’), shown in Figure 6 as a trivial rewrite that simply returns the original statement, and a robust compositional law (‘$fn_rewrite_compose$’), illustrated in Figure 8, we establish a framework where proofs of correctness for complex, composed rewrites are automatically derived from the proofs of their constituent parts. This inherent automation, stemming from the algebra's formal properties, greatly enhances the ergonomics and development speed for new optimisations. To further assist rewrite authors and allow them to focus on local transformations, we have formally defined rules for rewrites that apply directly to statements (‘$stmt_rewrite$’), along with a proven mechanism to lift these statement-level rewrites to function-level rewrites, ensuring their correctness propagates. A key higher-order rewrite, ‘$find_stmt_rewrite$’, depicted in Figure 9, automates the otherwise tedious process of recursively pattern matching and searching the AST for applicable nodes. This crucial abstraction means individual rewrite definitions only need to consider transforming a single AST node, greatly simplifying their proof obligations by abstracting away the complexity of AST traversal and application.

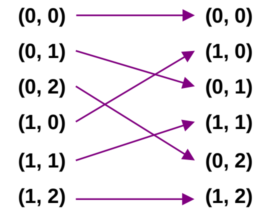

Fig. 7: An example iteration schedule change after a loop interchange.

D. Loop Optimisations

As a primary demonstration of *VLIM*'s capabilities, we focused on implementing loop interchange, a critical optimisation for improving cache performance in nested loops by enhancing spatial locality and cache line utilisation. Specifically, we targeted the interchange of two directly nested ‘for’ loops, transforming an ‘i-j’ iteration order to a ‘j-i’ order (conceptually shown as a change in iteration schedule in Figure 7). The permissibility of loop interchange is not arbitrary; it depends on whether the transformation will preserve the program's original behaviour, a principle known as **program semantics**. The foundational framework for this is **Bernstein's Conditions** [14], which formally define data dependencies between statements. While originally applied to the parallel execution of two distinct statements, these conditions are the basis for analysing dependencies between different *iterations* of a loop. The conditions identify three types of data hazards:

1) **Read-After-Write (RAW) Hazard (True Dependence):** This is the most common and intuitive dependency. An operation reads a value that was written by a previous operation. In a loop, this means an access in iteration k reads a value that was written in a previous iteration $k-1$. For example, `A[i] = ...; ... = A[i]`.
2) **Write-After-Read (WAR) Hazard (Anti-Dependence):** A subsequent operation overwrites a value that was previously read. For example, `... = A[i]; A[i] = ...`.

3) **Write-After-Write (WAW) Hazard (Output Dependence):** Two operations attempt to write to the same memory location. For example, `A[i] = ...; A[i] = ...`.

For loop interchange to be valid, it must preserve program semantics which requires a more sophisticated analysis beyond simple statement independence, as it must account for data dependencies carried across loop iterations. For instance, a dependency from iteration (i, j) to $(i+1, j)$ is carried by the outer loop and remains valid after an interchange. In contrast, a dependency from (i, j) to $(i-1, j+1)$ would likely invalidate the transformation. Hence, one should analyse the direction and distance of these dependencies to ensure that performance optimisation does not introduce a correctness bug.

```
Section Composition.
  Variable r1 : fn_rewrite.
  Variable r2 : fn_rewrite.

  Definition fn_rewrite_compose' f :=
    do f' <- r1.(rewrite) f; r2.(rewrite) f'.

  Lemma fn_rewrite_compose_correct :
    fn_rewrite_correct fn_rewrite_compose'.
  Proof.
  rewrite/fn_rewrite_compose' =>>.
  destruct r1.(rewrite) eqn:fr1 => //=.
  destruct r2.(rewrite) eqn:fr2 => //.
  injection 1 as <- =>> ef.
  move: (r1.(correct) _ _ fr1 _ _ _ _ ef) =>
       [] > [] ef' ->.
  move: (r2.(correct) _ _ fr2 _ _ _ _ ef') =>
       [] oe [] ? ->.
  by exists oe.
  Qed.

  Definition fn_rewrite_compose := {|
    rewrite := fn_rewrite_compose';
    correct := fn_rewrite_compose_correct;
  |}.
End Composition.
```

Fig. 8: A higher order rewrite that composes two rewrites and automatically derives their proof of correctness.

```
Fixpoint find_stmt_rewrite' s :=
  match r.(rewrite) s with
  | OK s' => OK s'
  | Error _ =>
    match s with
    | Sseq s1 s2 =>
      match find_stmt_rewrite' s1 with
      | OK s1' => OK (Sseq s1' s2)
      | Error _ => do s2' <- find_stmt_rewrite'
          s2; OK (Sseq s1 s2')
      end
    | Sifthenelse cond s1 s2 =>
      match find_stmt_rewrite' s1 with
      | OK s1' => OK (Sifthenelse cond s1' s2)
      | Error _ => do s2' <- find_stmt_rewrite'
          s2; OK (Sifthenelse cond s1 s2')
      end
    | Sloop s1 => do s1' <- find_stmt_rewrite'
          s1; OK (Sloop s1')
    | _ => Error [(MSG "Couldn't find a valid
                  application location.")]
    end
  end.
```

Fig. 9: A higher order rewrite that tries to apply a rewrite in a pre-order traversal of the AST until it successfully applies.

However, our modification relaxes the strict WAW requirement. While the original Bernstein condition [14] demands

```
Sloop (
  Sifthenelse
    (Ebinop_cmpu Oltu OInt64 (Eacc ("i", []))
       (Eacc (S 12 "_i_hi", [])))
    (Sseq
      (<EXPR>)
      (Sassign
        ("i", [])
        (Ebinop_arith Oadd OInt64 (Eacc
           ("i", []))
          (Econst (Cint64 Unsigned
             (Int64.repr 1))))))
    (Sbreak))
```

Fig. 10: The internal AST of a generic for loop.

disjoint memory cells for statements to commute, our approach allows commutation even when statements modify the same memory cell, provided that these modifications are themselves commutative. Formally, for any shared memory location $m \in W(S_1) \cap W(S_2)$, the operations S_1 and S_2 on m must satisfy $S_1(S_2(m_{initial})) = S_2(S_1(m_{initial}))$. This extension is vital for numerical algorithms, enabling the interchange of associative operations such as increments (e.g., '$i++$' or '$x+ = f(j)$'), which would otherwise violate the stricter condition. For instance, consider two statements: '$S1 : sum = sum + A[i]$' and '$S2 : sum = sum + B[j]$'. The original Bernstein condition would prohibit their interchange due to the shared write to 'sum'. Our modified condition, however, permits this because addition is commutative ($m_1 + m_2 = m_2 + m_1$). This allows for the interchange of associative operations like increments (e.g., '$i++$' or '$x+ = f(j)$') or array accumulation like '$sum = sum + data[i][j]$' , which would otherwise violate the stricter condition but preserve semantics due to operand associativity and commutativity. This formally-verified property validates loop interchange under our modified condition. The Capla AST represents all looping constructs as an 'Sloop' containing an 'Sifthenelse' block for loop bounds and an 'Sassign' for increment. This structure, visually depicted in Figure 10, guided our rewrite implementation, enabling effective pattern matching and operation on this abstracted form. Furthermore, the rewrite guard incorporates an inductive argument to formally handle loop bodies of runtime-known lengths, extending beyond simple two-statement commutation. This verified loop interchange serves as a foundational optimisation, with its correctness proof automatically composed within our algebra.

```
Definition apply_n :=
    Nat.iter n (stmt_rewrite_compose r)
    stmt_rewrite_id.
```

Fig. 11: A higher order rewrite that applies a rewrite 'n' times as well as its proof of correctness.

IV. Evaluation

This section evaluates *VLIM*'s verified loop interchange optimisation, detailing its impact on matrix multiplication performance and the practical aspects of our rewrite algebra through experimental setup, runtime, and cache miss analysis.

A. Simulation Setup and Configuration

Our experimental evaluation was conducted on a system featuring an 11th Gen Intel i5-1135G7 processor operating

at 2.6GHz. To ensure reliable and consistent measurements, each benchmark run for matrix multiplication was executed thousands of times, preceded by warm-up runs to minimise the cold-cache effects. CPU performance counters were employed to accurately compute cache misses, while a precise timer was utilised to record the overall runtime.

We assessed the performance across three distinct compilation choices and various iteration orders:

- **Compcert Runs:** Programs were compiled directly from Capla source code to machine code using the Compcert verified compiler. These represent the baseline for formally verified compilation paths.
- **Clang Runs:** Capla source code was first compiled to C via Compcert, and subsequently compiled to machine code using the Clang compiler, which was configured with more aggressive optimisations.
- **Clang + Polly:** For state-of-the-art comparison, the Clang compilation was augmented with Polly, a polyhedral optimiser. Polly is designed to freely rearrange and tile loops for optimal performance, though our current work does not yet support all optimisations Polly performs.

The iteration orders for the matrix multiplication loops were varied: 'ijk' (the straightforward default order), 'ikj' (a more cache-friendly interchange order), and 'freeform' for Polly, which dynamically determines its own optimal iteration pattern. This setup allowed us to directly observe the performance benefits of our verified 'ikj' loop interchange against both non-optimised and aggressively optimised baselines.

B. Performance Analysis

The experimental results, detailing average runtimes and cache misses for matrix multiplication across different matrix sizes and compilation choices, are presented in Table I and Table II, respectively. Table I clearly demonstrates the performance benefits of loop interchange, especially for larger matrix sizes. For a 1000×1000 matrix, the loop interchange ('ikj') implemented via *VLIM* significantly reduced runtime:

- Under Compcert, runtime decreased (by 36.6%) from 3.58s ('ijk') to 2.27s ('ikj').
- Under Clang, *VLIM*'s formally verified loop interchange notably reduced runtime by 74.6% (1.65s to 0.42s), offering critical assurances.

This highlights that even when using a highly optimising compiler like Clang, explicit, verified loop transformations can yield substantial speedups. The $Clang + Polly$ configuration, performing extensive polyhedral optimisations including tiling, achieved the best runtime at 0.265s for the 1000×1000 matrix. This superior performance demonstrates the capabilities of aggressive, unverified optimisation techniques and indicates potential for more gains beyond simple loop interchange, which our current verified framework does not yet encompass. The purpose of this comparison is to establish a high-performance target that future work on *VLIM* will aim to match, but with the added assurance of formal verification that $Clang + Polly$ lacks.

TABLE I: The average runtime, in seconds, of matrix multiplication at different matrix sizes for different iteration orders and compilers.

Compiler	**Iteration Order**	10×10	100×100	1000×1000
Compcert	ijk	0.000713	0.003820	3.581167
Compcert	ikj	0.000852	0.003432	2.268485
Clang	ijk	0.000887	0.001876	1.647675
Clang	ikj	0.000881	0.001404	0.423389
Clang + Polly	freeform	0.000835	0.001617	0.265363

TABLE II: The average number of cache misses of matrix multiplication at different matrix sizes for different iteration orders and compilers.

Compiler	**Iteration Order**	10×10	100×100	1000×1000
Compcert	ijk	21	90.4	29000000
Compcert	ikj	19	71.6	22300000
Clang	ijk	21.6	47.3	50800000
Clang	ikj	20.1	39.2	19700000
Clang + Polly	freeform	21.6	65.4	687000

Table II provides insight into the underlying cache behaviour. Cache misses become significantly more relevant with larger input sizes. For the 1000×1000 matrix:

- Clang with 'ikj' order showed a substantial reduction in cache misses (19.7 million) compared to 'ijk' (50.8 million). This directly correlates with its superior runtime performance, validating the cache-friendly nature of loop interchange. The substantial reduction in cache misses for the 'ikj' directly translates to an improved Misses Per Thousand Instructions (MPKI) and a higher Instructions Per Cycle (IPC), indicating more efficient processor utilisation due to better data locality, detailed analysis of which will be conducted in our future work.
- Interestingly, the Compcert's runs showed a smaller relative difference in cache misses between 'ijk' (29 million) and 'ikj' (22.3 million) despite a noticeable runtime improvement. This suggests that Compcert's internal optimisations or IR handling might not fully exploit the cache benefits for the 'ikj' order in the same way Clang does, or that other factors dominate its runtime.
- Notably, $Clang + Polly$ achieved drastically lower cache misses, reporting only $687,000$ for the 1000×1000 matrix. This is two orders of magnitude less than both Compcert and standard Clang, primarily attributed to its additional loop tiling optimisations that our current verified framework does not yet implement. This comparison underscores the critical need to extend verified frameworks to include more advanced techniques to bridge the performance gap with purely performance-driven tools, while retaining the formal guarantees that are paramount for high-assurance applications.

C. Discussion

VLIM demonstrates practical utility for verified loop optimisations, as its proof architecture streamlines composition and automated verification, enhancing efficiency. For instance, 'apply_n' (Figure 11) exemplifies trivial provability for sequenced rewrites, enabling semantic-preserving rule combinations. Though proving novel rewrites is non-trivial, the

framework guides robust composition from smaller proofs. High-level AST operations ensure user applicability, easy debugging, and fluid optimiser experimentation. Performance confirms significant speedups from verified loop interchange, particularly with Clang. Divergent cache improvements suggest Compcert's current internal heuristics may not fully exploit 'ikj' ordering's cache benefits like Clang. Crucially, while $Clang + Polly$ achieved vastly superior performance (two orders of magnitude fewer cache misses for 1000×1000 matrix) due to advanced loop tiling beyond our current verified framework, this highlights our initial AST-level transformations' limitations for aggressive polyhedral optimisations.

VLIM's *novelty stems from its formal correctness guarantees, a feature absent in purely performance-driven tools like Polly.* Integrating polyhedral theories for verified tiling will bridge the performance gap while maintaining formal guarantees. The modularity of rewrite algebra enables one to extend *VLIM* to complex optimisations such as tiling or unrolling via automatically composed local correctness proofs. Proof composition streamlines verified optimisation, yet novel rewrites demand deep, one-time formal reasoning. These proven rules, once automatically composed, significantly reduce the future verification burden. The process of formal verification requires towards the construction of mathematical proofs to demonstrate that the optimised code adheres to its intended behaviour for all inputs and scenarios. This is a computationally intensive task that requires the use of a theorem prover, in *VLIM*'s case, Rocq, to ensure the correctness of the rewrite rules and their compositions. Unlike unverified optimisations that rely on heuristics and empirical testing, *VLIM*'s approach requires the system to perform a deep, one-time formal reasoning for each new rewrite rule. However, once a rule is proven and incorporated into the rewrite algebra, its correctness proof can be automatically composed with other proven rules. This streamlines the verified optimisation process for future use and significantly reduces the future verification burden. Although proof generation incurs measurable compilation overhead initially, it is justified by high-assurance guarantees crucial for safety-critical applications.

V. Related Work

Existing research relevant to our work falls into two primary fields: verified compilation and loop-based cache optimisations. Our work builds on Capla, an alternative front-end for the pioneering Rocq-based Compcert verified C compiler [9], whose formal semantics seamlessly extend Compcert's semantic preservation proofs [10]. Verified optimisation efforts within Compcert include internal loop-invariant code motion [15] and peephole optimisations [13]. Other significant verified compilers, such as CakeML, offer high-assurance through bootstrapping [16]. Alternative approaches include OptiTrust, an OCaml framework employing "checked rewrites" with translation validation for C code [17], [18], and externalised optimisation passes (e.g., by Ikarashi et al.) for fine-grained, error-resistant manual rewrite application. Translation validation is also used by systems like the seL4 microkernel to verify highly optimised binaries [19]. Many such efforts focus on internal compiler passes or post-optimisation validation, generally lacking a framework for user-applied transformations with automatic proofs, which *VLIM* provides. While recent methods use functional iteration representations [20], *VLIM* operates on imperative code using composable higher-order rewrites. This approach enables generalization beyond loops to other program transformations.

Loop cache optimisations, critical for memory access, grow intricately with nested loops and complex access [14]. Convex integer polyhedral optimisation models iteration spaces for optimal loop rescheduling patterns. While solvers offer theoretical optimality, manual control can provide practical advantages. Shared cache utilisation optimisation began with Wolf et al. [21]. Recent work includes frameworks combining software cache partitioning, loop tiling, and data layout transformations [22], [23], defensive tiling [24], and tile size selection models [25], including "peer-aware" tiling for DNNs [26]. The growing intersection of formal verification and polyhedral models includes verified theories of convex polyhedra [27] and formally verified code generators for the polyhedral model [28]. However, these performance-focused approaches often lack formal correctness guarantees, hindering their deployment in high-assurance environments.

Unlike prior work separating performance optimisation from formal verification, *VLIM synthesises a unique rewrite algebra for user-applied, cache-aware loop transformations with automatic correctness proofs.* While OptiTrust offers checked rewrites [17], *VLIM provides a unified, provably correct approach ensuring semantic preservation for complex loop transformations—a guarantee largely absent in performance-focused, manual optimisation.* This enables highly optimised, formally verified performance-critical code.

VI. Conclusions and Future Work

This paper presented *VLIM*, a novel rewrite algebra for verified loop optimisations in the Capla programming language. *VLIM*'s most significant contribution lies in the automation of correctness proofs for composed rewrites, a feature that provides strong guarantees of semantic preservation throughout the optimisation process. By integrating this formal verification directly into the optimisation pipeline, *VLIM* bridges the historical gap between performance and correctness in compiler design. A major finding is the significant performance improvement achieved by verified loop interchange relative to its proof burden: for a 1000×1000 matrix, runtime was reduced by 36.6% (from 3.58s to 2.27s) and 74.6% (from 1.65s to 0.42s) while compiled with Compcert and Clang, respectively. These findings not only demonstrate the efficacy of our method in generating high-performing code but also underscore the value of a provably correct optimisation, offering a powerful tool for developing high-assurance systems where both speed and reliability are paramount. We aim to match Polly's performance by integrating polyhedral-verified tiling and exploring runtime-based power/performance optimisations.

REFERENCES

[1] N. R. Mahapatra and B. Venkatrao, "The processor-memory bottleneck: problems and solutions," *XRDS*, 1999.

[2] J. Sebot and N. Drach-Temam, "Memory bandwidth: The true bottleneck of simd multimedia performance on a superscalar processor," in *Euro-Par*, 2001.

[3] R. Fabian, *Data-oriented design: software engineering for limited resources and short schedules*, 2018.

[4] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.

[5] R. N. Watson, J. Baldwin, D. Chisnall, T. Chen, J. Clarke, B. Davis, N. Filardo, B. Gutstein, G. Jenkinson, B. Laurie, A. Mazzinghi, S. Moore, P. G. Neumann, H. Okhravi, A. Richardson, A. Rebert, P. Sewell, L. Tratt, M. Vijayaraghavan, H. Vincent, and K. Witaszczyk, "It is time to standardize principles and practices for software memory safety," Tech. Rep., 2025.

[6] I. Jovanović, "Software testing methods and techniques," *The IPSI BgD transactions on internet research*, 2006.

[7] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods*, 2011.

[8] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, 2015.

[9] G. Melquiond and J. Moreau, "A safe low-level language for computer algebra and its formally verified compiler," *Proc. ACM Program. Lang.*, 2004.

[10] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, 2009.

[11] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[12] K. Ji, M. Ling, L. Shi, and J. Pan, "An Analytical Cache Performance Evaluation Framework for Embedded Out-of-Order Processors Using Software Characteristics," *ACM TECS*, 2018.

[13] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for compcert," in *PLDI*, 2016.

[14] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. on Electronic Computers*, 1966.

[15] D. Monniaux and C. Six, "Formally verified loop-invariant code motion and assorted optimizations," *ACM TECS*, 2022.

[16] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified cakeml compiler backend," *Journal of Functional Programming*, 2019.

[17] A. Charguéraud, B. Bytyqi, D. Rouhling, and Y. Barsamian, "OptiTrust: An Interactive Framework for Source-to-Source Transformations," 2022. [Online]. Available: https://inria.hal.science/hal-03773485

[18] G. Bertholon, A. Charguéraud, T. Kundefinedhler, B. Bytyqi, and D. Rouhling, "Interactive source-to-source optimizations validated using static resource analysis," in *SOAP*, 2024.

[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM TOCS*, 2014.

[20] A. Liu, G. L. Bernstein, A. Chlipala, and J. Ragan-Kelley, "Verified tensor-program optimization via high-level scheduling rewrites," *Proc. ACM Program. Lang.*, no. POPL, 2022.

[21] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *SIGPLAN Not.*, 1991.

[22] K. Vasilios, K. Georgios, and V. Nikolaos, "Combining software cache partitioning and loop tiling for effective shared cache management," *ACM TECS*, 2018.

[23] V. Kelefouras, G. Keramidas, and N. Voros, "Cache partitioning + loop tiling: A methodology for effective shared cache management," in *ISVLSI*, 2017.

[24] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *CGO*, 2013.

[25] K. Narasimhan, A. Acharya, A. Baid, and U. Bondhugula, "A practical tile size selection model for affine loop nests," in *ICS*, 2021.

[26] J. Zhao, H. Cui, Y. Zhang, J. Xue, and X. Feng, "Revisiting loop tiling for datacenters: Live and let live," in *ICS*, 2018.

[27] X. Li, H. Liang, and X. Feng, "Verified validation for affine scheduling in polyhedral compilation," in *TASE*, 2024.

[28] N. Courant and X. Leroy, "Verified code generation for the polyhedral model," *Proc. ACM Program. Lang.*, 2021.