

Efficient Compilation in Compilers through Uncovering and Eliminating Redundancies

O. O. Omonijo¹, A. O. Olateju², O. O. Adesanya³, C. Ebirika⁴, A. T. John-Dewole⁵

^{1,2,4} Computer Science Unit, Nigeria Maritime University, Okerenkoko, Delta State, Nigeria

³Department of Computer Science, Federal Cooperative College, Ibadan, Oyo State, Nigeria

⁵Department of Computer Science, Lead City University, Ibadan, Oyo State, Nigeria.

Article Info

Article History:

Published: 30 Dec 2025

Publication Issue:

Volume 2, Issue 12
December-2025

Page Number:

1115-1126

Corresponding Author:

O. O. Omonijo

Abstract:

Redundancy elimination is aimed at reducing unnecessary work and improve execution efficiency of compilers. The efficiency of redundancy elimination has become increasingly important as modern software systems increase in size and complexity, thereby, influencing compilation scalability as well as runtime performance. Redundancy elimination techniques are still frequently applied as isolated optimizations despite extensive prior work, with little emphasis on their coordination, cost, and interaction effects within modern compilation pipelines. Although methods like value numbering, Loop-Invariant Code Motion (LICM), Common Subexpression Elimination (CSE), and Partial Redundancy Elimination (PRE) are thoroughly covered in literature, integrative analyses that examine how these methods interact with one another and with downstream optimizations are still lacking. Specifically, there is insufficient synthesis of how redundancy elimination affects loop optimization, register allocation, and instruction scheduling in modern compiler infrastructures. The objective of this study is to present an integrated and critical examination of redundancy elimination in modern compilers, focusing on trade-offs, interactions, and design considerations rather than introducing new algorithms. Using Static Single Assignment (SSA) based intermediate representations as a unifying foundation, the study employs a table-driven analytical evaluation of redundancy categories and investigates their integration with loop optimization and code motion techniques. The analysis shows that while eliminating redundancies can significantly reduce duplicated computation, aggressive or poorly coordinated application may result in increased register pressure, disrupted vectorization patterns, and reduced Instruction-Level Parallelism (ILP). These findings highlight the need for coordination-sensitive and cost-conscious optimization techniques. This study offers insights that inform practical compiler design as well as compiler research by redefining redundancy elimination as an interacting optimization system.

Keywords: redundancy elimination, compiler optimization, partial redundancy elimination, loop optimization, code motion

1. Introduction

In the design of contemporary software systems, compilers are expected to produce high-performance executable code while working under increasingly complex constraints hence, the need for efficient compilation. The cost of unnecessary computing becomes pronounced with the growth

and complexity of programs, including embedded systems, mobile platforms, data-intensive workloads, and machine-learning-driven applications. One of the very effective methods for improving compiled code efficiency is by eliminating redundancy. This method identifies and remove computations that are repeated without contributing new information to program execution (Aho et al., 2006; Muchnick, 1997).

Repeated arithmetic expressions, loop-invariant computations, partially redundant expressions along different control-flow paths, unnecessary value re-computations, and dead or unreachable code are only a few examples of how redundancies appear in programs. Such redundancies, if left unaddressed, wastes processor and memory resources, lengthen execution times, and increase code size. As a result, redundancy elimination has been a major focus of compiler optimization research for a long time. It has given rise to a wide range of techniques such as Common Subexpression Elimination (CSE), Loop-Invariant Code Motion (LICM), Partial Redundancy Elimination (PRE), Value Numbering, and Dead Code Elimination (DCE) (Rosen et al., 1988; Dasgupta, 1991; Cooper & Torczon, 2012).

Historically, most of these strategies were created separately, with each of them focusing on a specific class of redundancy under particular program structures. Even if isolated methods work well in their own domains, they frequently fail to expose deeper or cross-cutting redundancies that arise from complex control flow, nested loops, or interactions between multiple optimization steps. This observation inspired the development of PRE method. PRE combines elements of CSE and LICM to remove expressions that are redundant along some but not all execution pathways (Morel & Renvoise, 1979; Dasgupta, 2002). PRE and its variations continue to be among the most effective redundancy elimination techniques used in modern compilers, most especially when developed over Static Single Assignment (SSA) form, which simplifies dominance, equivalency, and data-flow reasoning, (Cytron et al., 1991; Briggs et al., 1998; Chow, 2021).

Redundancy elimination techniques present nontrivial problems despite their effectiveness. Advanced optimization techniques such as PRE and Global Value Numbering (GVN) can significantly increase compilation time, register pressure, and code motion complexity. Additionally, according to recent studies, aggressively optimized code might paradoxically reduce overall compiler effectiveness by interfering with subsequent optimization passes or exposing inefficiencies in cost models (Cooper et al., 2008; Gao et al., 2024). These results highlight an important reality: especially in modern compilation pipelines, removing more redundancy does not always lead to better performance.

With the emergence of new application domains, redundancy elimination has become even more relevant. Code generated by high-level modelling environments like Simulink, data-parallel and graph-based workloads, mobile and Android applications, and even machine-generated or neural code must be optimized by modern compilers (Yu et al., 2024; Wang et al., 2022; Zhang et al., 2025; Jiang et al., 2025). Redundancies in these contexts frequently arises from recurring data-flow patterns, structural similarities, domain-specific abstractions, and conventional scalar expressions. Hence, integrated approaches, whole-program analysis, and rigorous trade-offs between optimization benefit and compilation cost are becoming increasingly important in modern redundancy elimination (Roy & Paleri, 2024; Cai, 2025).

In order to address the needs of modern compilation, this paper explores redundancy elimination from both a foundational and modern standpoint, emphasizing how traditional techniques can be expanded, merged, and transformed. Rather than treating redundancy elimination methods as independent optimizations, the study highlights the relationships, limitations, and integration of redundancy removal techniques. This includes the function of code reshaping and associativity transformations in exposing previously undetected redundancies. This paper attempts to clarify when and how redundancy elimination contributes most effectively to efficient compilation by critically analyzing both established techniques and recent advances.

2. Literature Review

Redundancy elimination research has developed along with compiler theory and practice, thereby reflecting shifts in application domains, hardware architectures, and programming paradigms. The theoretical foundations for finding redundancies in computations were laid by early research, and these foundations were further improved through program representations, data-flow analysis, and integration with other optimization techniques. Redundancy elimination has been reviewed in more recent studies by examining contemporary workloads, compiler infrastructures, and efficiency constraints.

2.1 Foundational Work on Redundancy Elimination

The earliest systematic approaches to elimination of redundancy focused on identifying equivalent computations within programs and avoiding their repeated evaluation. Rosen et al. (1988) introduced a ground breaking method which is referred to as Global Value Numbering (GVN). It is a seminal approach that assigns value numbers to expressions in order to effectively identify redundant calculations. Value equivalency was developed by this study as a fundamental concept in redundancy elimination, and it continues to have an impact on modern compilers.

Subsequent foundational texts and studies, such as Muchnick (1997) and Aho et al. (2006), consolidated redundancy elimination as a central component of compiler optimization. These works formalized techniques including CSE, DCE, and LICM, emphasizing their role in reducing execution time and code size. However, early approaches were often limited in scope, typically operating locally within basic blocks or under restrictive control-flow assumptions.

2.2 Partial Redundancy Elimination and Data-Flow-Based Approaches

A significant breakthrough in redundancy elimination was the development of Partial Redundancy Elimination (PRE) by Morel and Renvoise (1979), which Dasgupta (1991) later modified and improved. PRE unified CSE and LICM together by enabling the removal of redundant expressions along some, but not all execution paths. This marked a significant shift from purely global or solely local optimizations to a more complex, path-sensitive analysis.

The goal of PRE's further refinements was to make its formulation simpler and increase its practical applicability. Hence, the E-path-PRE, which reduces algorithmic complexity while preserving optimization capability, was proposed by Dasgupta (2002). Subsequently, different trade-offs between aggression and safety were explored by alternative formulations including value-based PRE and lazy code motion (Van Drunen & Hosking, 2004; Dasgupta & Gangwani, 2019). These studies demonstrated that, although often at the expense of higher compilation overhead, PRE could be adapted to different intermediate representations and optimization goals.

2.3 Static Single Assignment Form and Redundancy Elimination

Compiler optimization was fundamentally reshaped with the introduction of the Static Single Assignment (SSA) form. Cytron et al. (1991) demonstrated how SSA makes dominance and equivalency relationships explicit by guaranteeing that every variable is assigned exactly once, thereby simplifying data-flow analysis. Briggs et al. (1998) built on this by showing how to generate and destroy SSA more practically, as a result, made it possible for production compilers to use it widely.

SSA-based formulations of redundancy elimination have shown particular efficacy. Chow (2021) offers a thorough explanation of redundancy elimination strategies in SSA-based compiler design, demonstrating how SSA reduces analysis complexity to support PRE, GVN, and LICM.

However, redundancy removal is still difficult even within SSA, especially when dealing with complex control-flow graphs or integrating with other optimizations.

2.4 Reassessment of Redundancy Elimination Efficiency

With the growth in compiler infrastructures, researchers began to critically reevaluate redundancy elimination techniques instead of only extending them. Redundancy elimination was reviewed by Cooper et al. (2008), who highlighted cases in which aggressive optimizations resulted in decreasing returns or obstruct subsequent compilation phases. This line of work highlighted the need to assess redundancy elimination based on its effect on compilation quality as well as the quantity of code eliminated.

Gao et al. (2024) challenged the notion that more aggressive redundancy elimination always results in improved performance by demonstrating that code optimized for efficiency at one step may induce inefficiencies later in the optimization pipeline. Similarly, Roy and Palleri (2024) used iterative data-flow analyses to examine PRE and found trade-offs between computational cost and optimization efficacy. These results highlight the necessity of context-sensitive and cost-aware redundancy elimination techniques.

2.5 Redundancy Elimination in Modern and Emerging Domains

Redundancy elimination now expands beyond traditional scalar and loop-based programs. Redundancy removal in code generation for data-intensive Simulink models was investigated by Yu et al. (2024), highlighting its significance in model-based design environments. Wang et al. (2022) examined compiler feedback mechanisms for neural code generation concurrently, highlighting new forms of redundancy introduced by machine-generated programs.

Emerging workloads such as graph neural network training (Zhang et al., 2025), mobile and Android applications (Jiang et al., 2025), and large-scale storage and cloud systems (Yu, 2022) have also benefited from redundancy elimination. Multiple optimization techniques and more comprehensive program analyses are motivated by these domains, which present structural, data-flow, and parallel redundancies that are not fully addressed by conventional techniques.

2.6 Synthesis and Research Gap

Redundancy elimination has advanced from isolated, local optimizations to complex, SSA-based, and integrated techniques, as observed in the reviewed literatures. Although traditional techniques like value numbering, CSE, LICM, and PRE remain fundamental, their limitations are becoming more noticeable in modern compilation activities. Recent studies emphasize the significance of balancing the advantages of optimization against compilation costs and the necessity of taking interactions between optimization passes into account.

Despite a great deal of earlier work, there is still a lack of knowledge regarding the most effective ways to combine redundancy elimination techniques, reshape code to expose hidden redundancies, and modify traditional methodologies to modern workloads. In order to overcome these challenges, existing approaches must be expanded upon and their efficacy in modern compiler pipelines must be critically assessed. This paper examines redundancy elimination techniques from both foundational and modern perspectives, with a focus on integration, trade-offs, and practical applicability.

3. Analysis of Redundancy Elimination Frameworks in Compiler Optimization

Redundancy in programs can appear at many structural levels and execution scopes. These can range from value-equivalent computations across complex control-flow paths to repeated expressions within a basic block. Over time, compiler research has developed numerous redundancy elimination techniques, each suited to certain redundancy patterns and program representations. While more sophisticated methods like PRE and GVN handle path-sensitive and value-based redundancies (Rosen et al., 1988; Dasgupta, 1991; Van Drunen & Hosking, 2004), Classical techniques like CSE and LICM remain fundamental (Aho et al., 2006; Muchnick, 1997).

As illustrated in Figure 1, redundancy elimination in modern compilers operates over an SSA-based intermediate representation and is best understood as a layered process. Rather than applying individual optimizations in isolation, redundancy detection techniques such as CSE, value numbering, and PRE are feed into an integration and coordination layer responsible for code motion and profitability decisions. This integration-centric view emphasizes that redundancy elimination is most effective when treated as an interacting system rather than a sequence of independent transformations. This integrated perspective provides the structural context for the redundancy categories discussed in this study.

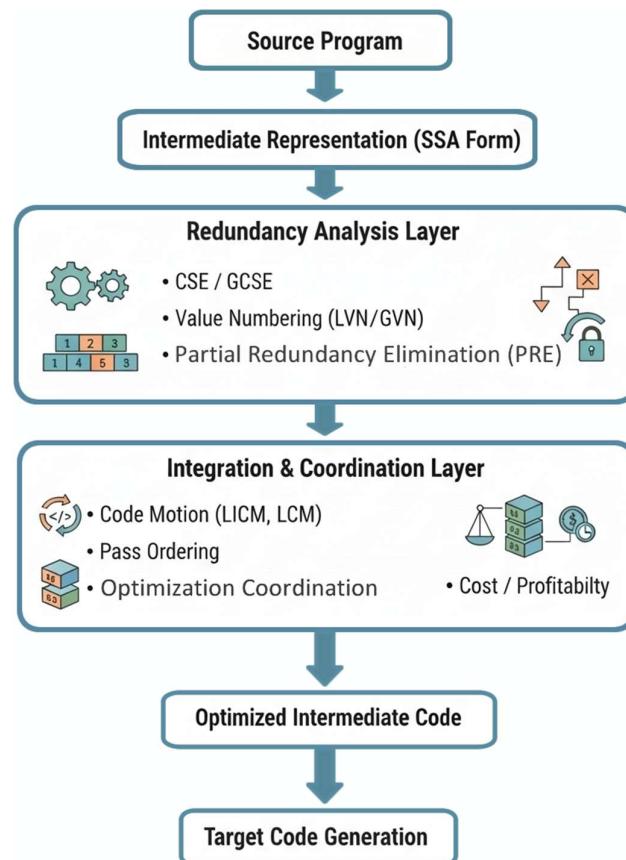


Figure 1. Integrated Redundancy Elimination Pipeline in Modern Compilers

Table 1 presents a comparative analysis of the major redundancy categories and the techniques used to eliminate them, highlighting their scope, strengths, limitations, and relevance to modern compilation.

Table 1: Analytical Comparison of Redundancy Categories and Elimination Techniques

Redundancy Category	Typical Source	Primary Elimination Technique(s)	Scope of Analysis	Strengths	Limitations / Trade-offs	Study References
Common Subexpression Redundancy	Repeated evaluation of identical expressions with unchanged operands	Common Subexpression Elimination (CSE)	Local Common Subexpression Elimination (LCSE) or Global Common Subexpression Elimination (GCSE)	Reduces duplicate computations; low overhead for LCSE; effective for straight-line code	GCSE requires control-flow analysis; limited for path-sensitive or semantically equivalent expressions	Aho et al. (2006); Muchnick (1997); Enyindah & Okon (2017)
Loop-Invariant Redundancy	Expressions inside loops whose values do not change across iterations	Loop-Invariant Code Motion (LICM)	Loop and dominance region	Significant runtime improvement for loop-intensive programs; enables further optimizations	Requires dependence and alias analysis; may increase register pressure	Muchnick (1997); Cooper & Torczon (2012); Ding & Shen (2017)
Partial Redundancy	Expressions redundant along some, but not all, execution paths	Partial Redundancy Elimination (PRE); Lazy Code Motion	Global, path-sensitive	Eliminates redundancies missed by CSE and LICM; unifies multiple optimizations	High analysis cost; may increase compilation time; interacts with other optimizations	Dasgupta (1991, 2002); Van Drunen & Hosking (2004); Roy & Palleri (2024)
Value-Based Redundancy	Different expressions computing equivalent values	Local and Global Value Numbering (LVN, GVN)	Basic block or whole program	Efficient detection of equivalence; simplifies redundancy checks	Traditional GVN focuses on syntactic equality; limited for complex control flow	Rosen et al. (1988); Cooper et al. (2008); Nabizath & Vineeth (2016)
Dead and Unreachable Code	Computations whose results are	Dead Code Elimination (DCE)	Local and global	Reduces code size; enables	Does not directly reduce	Jagdish (2014);

	unused or code paths never executed			further optimizations	computation unless combined with other passes	Aho et al. (2006)
Structural / Shape-Induced Redundancy	Rigid expression associativity and CFG structure hiding equivalence	Code Reshaping; Algebraic Transformations; Integrated PRE + GVN	Global, representation-dependent	Exposes hidden redundancies; increases optimization opportunities	Risk of optimization interference; may introduce new redundancies	Cooper et al. (2008); Cooper & Torczon (2012); Cai (2025)
Domain-Specific Redundancy	Repeated data-flow patterns in specialized workloads	Integrated redundancy elimination frameworks	Whole-program, domain-aware	Effective for modern workloads (ML, Simulink, mobile)	Requires domain knowledge; higher analysis complexity	Yu et al. (2024); Wang et al. (2022); Zhang et al. (2025)

Within the redundancy analysis layer shown in Figure 1, different categories of redundancy exhibit varying scopes and analytical requirements, thereby, motivating the comparative framework presented in Table 1. As observed from the comparative analysis in Table 1, no single redundancy elimination technique is sufficient to address all redundancy patterns encountered in modern programs. While methods like CSE and LICM work well for loop-centric and localized redundancies, they are unable to identify path-sensitive or value-based equivalencies. These limitations were addressed by PRE and GVN, but they incurred a greater computational cost and complex interactions with other optimization passes. Furthermore, it is also observed that the effectiveness of redundancy elimination is strongly influenced by program representation. SSA-based formulations facilitate more precise redundancy detection by streamlining dominance analysis and equivalency reasoning. However, even within SSA, aggressive redundancy elimination may negatively impact downstream optimizations, highlighting the importance of cost-aware application strategies. Lastly, integrated and domain-aware approaches are motivated by the fact that data-intensive models, neural code generation, and graph-based workloads frequently exhibit structural or semantic redundancy that are difficult to capture by conventional techniques.

4. Loop Optimization and Integration with Redundancy Elimination

Loops are among the most performance-critical structures in modern programs, majorly because they often dominate execution time, especially in scientific computing, data-intensive applications, and machine-learning workloads. As a result, there have been a strong emphasis on loop optimization in compiler research for a long time. The practical impact of many redundancy elimination techniques, like LICM, PRE, and value-based optimizations, is largely dependent on how effectively they work with loop transformations (Aho et al., 2006).

4.1 Loop-Centric Sources of Redundancy

Redundancies within loops are mainly caused by redundant memory accesses, equivalent computations performed along different loop paths, or repetitive evaluation of expressions whose

operands stay unchanged throughout iterations. These redundancies are especially expensive because their execution frequency increases with the number of loop iterations. LICM directly addresses this problem and minimizes redundant processing by finding invariant expressions and moving them outside the loop body (Muchnick, 1997).

However, loop structures often interact with control flow in complex ways. Partial redundancy that LICM cannot remove on its own may result from conditional branching inside loops. PRE becomes essential in these situations because it may minimize redundant evaluations by hoisting or sinking calculations across loop boundaries and conditional paths (Van Drunen & Hosking, 2004).

4.2 Integration of PRE, Value Numbering, and Loop Transformations

Effective loop optimization rarely relies on a single technique. Instead, modern compilers integrate multiple redundancy elimination techniques so as to maximize optimization opportunities. Value numbering methods make it possible to identify equivalency of expressions computed within and across loop iterations, especially when paired with SSA form (Rosen et al., 1988; Briggs et al., 1998). By identifying safe and profitable code motion opportunities, these equivalencies can then guide LICM and PRE.

Hybrid approaches that combine value numbering, PRE, and scalar replacement have been shown to expose redundancies that are otherwise hidden by rigid code structure (Cooper et al., 2008). Compilers can identify invariant or partially redundant computations that were not apparent in the original program form by changing associativity and restructuring expressions. Such integration is particularly important for loops with array accesses and complex indexing expressions, where redundancies may span multiple iterations.

4.3 Interaction with Other Loop Optimizations

Other loop optimizations, such as loop unrolling, loop fusion, loop vectorization, and loop tiling, are closely related to redundancy elimination within loops. For instance, by duplicating loop bodies, loop unrolling can expose additional opportunities for CSE and value numbering, while loop fusion can eliminate cross-loop redundancy by combining adjacent loops with similar structures (Cooper & Torczon, 2012).

Conversely, extensive redundancy removal can negatively affect subsequent loop optimizations. Excessive code motion might complicate memory access patterns, impede vectorization, or increase register pressure. This emphasizes the necessity of carefully ordering and coordinating optimization passes so that redundancy elimination complements rather than conflicts with loop-level transformations.

4.4 Redundancy Elimination in Modern Loop-Intensive Workloads

Recent application domains further emphasize the importance of integrated loop optimization. Deeply nested loops with structurally similar computations are frequently found in data-intensive models produced by high-level environments like Simulink (Yu et al., 2024). According to Zhang et al. (2025) and Wang et al. (2022), graph-based and machine-learning workloads exhibit repetitive data-flow patterns that benefit from combined redundancy elimination and loop transformations. These emerging workloads motivate the use of cost-aware and domain-sensitive optimization strategies, where redundancy elimination is applied selectively based on expected performance gains. As examined by Roy and Paleri (2024), one such attempt to strike a balance between optimization efficacy and compilation overhead in loop-intensive environments is iterative data-flow analysis for PRE.

5. Discussion

Redundancy elimination has long been regarded as a fundamental component of compiler optimization. However, as compiler infrastructures and target workloads have evolved, it has become increasingly clear that redundancy elimination is not a universally beneficial transformation. Rather, it is a complex combination of trade-offs involving compilation cost, interaction with other optimizations, and sensitivity to contemporary program characteristics determine how effective it is.

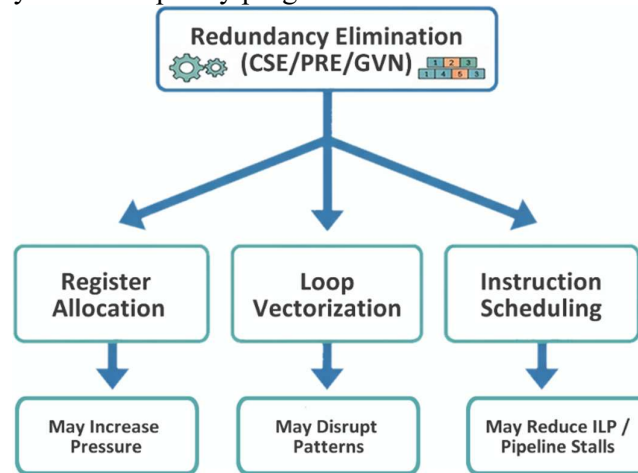


Figure 2. Interactions and Trade-Offs Between Redundancy Elimination and Downstream Optimizations

While redundancy elimination reduces duplicated computation, building on the integrated pipeline shown in Figure 1, Figure 2 highlights how aggressive or poorly coordinated redundancy elimination can interact negatively with register allocation, loop vectorization, and instruction scheduling, reinforcing the need for cost-aware and coordination-sensitive optimization strategies. As highlighted by recent studies, optimized code may unintentionally increase register pressure or disrupt optimization-friendly program structure, underscoring the need for cost-aware and interaction-sensitive redundancy elimination strategies.

One of the central trade-offs in redundancy elimination lies between the quality of optimized code and the cost of performing the analysis itself. Techniques such as LCSE and basic DCE are inexpensive and reliably beneficial, making them attractive in both ahead-of-time and just-in-time compilation settings. In contrast, more powerful techniques like PRE and GVN both require sophisticated data-flow analysis, dominance reasoning, and often iterative fixed-point computation.

As demonstrated by recent work by Roy and Palleri (2024), iterative data-flow formulations of PRE, while precise, can incur significant computational overhead. In large or complex programs, the marginal improvements gains achieved by eliminating additional redundancies may not justify the increased compilation time. This trade-off is especially important in environments where fast compilation is essential, such as interactive development tools and dynamic language runtimes.

According to growing body of research, redundancy elimination cannot be evaluated in isolation. Gao et al. (2024) provide compelling evidence that code optimized for efficiency at one stage of the compilation pipeline can introduce inefficiencies at later stages. For example, aggressive elimination of redundant computations may increase register pressure, disrupt instruction scheduling, or inhibit loop vectorization, ultimately degrading runtime performance despite reducing apparent redundancy. Similarly, Cooper et al. (2008) demonstrated that when redundancy elimination is used without careful coordination, it may interfere with other optimizations. Code motion associated with PRE or LICM can obscure patterns relied upon by subsequent passes, such as memory access regularity or vectorization-friendly loop structures. These results challenge the traditional assumption that

redundancy elimination is always advantageous and emphasize the importance of pass ordering and integration.

Another subtle challenge is the introduction of secondary redundancies. Code-resaping transformations, including changing expression associativity or hoisting computations, may unintentionally create new redundant expressions or expose dead code that was not previously undiscovered. As a result, there may be need to reapply redundancy elimination passes, which would increase pipeline complexity and compilation time. Cooper and Torczon (2012) note that such aggressive optimization pipelines have these kinds of feedback effects, where one transformation either enables or inhibits others. As was earlier pointed out, code reshaping can expose hidden redundancies, but in order to avoid counterproductive transformations, it must be guided by accurate cost models.

The redundancy patterns introduced by modern application domains are substantially different from the patterns assumed by traditional compiler optimizations. High-level structural or semantic redundancy that is not easily captured by traditional syntactic analyses is frequently present in machine-generated code, data-intensive models, and graph-based workloads (Wang et al., 2022; Yu et al., 2024; Zhang et al., 2025). In such cases, eliminating redundancies might require domain-aware or whole-program analyses, increasing both complexity and uncertainty. These workloads frequently rely on deeply nested loops, parallel execution, and irregular memory access patterns. In some situations, aggressive redundancy reduction can worsen performance bottlenecks or interfere with hardware-level optimizations, which emphasizes the need for selective and adaptive approaches.

The challenges discussed above suggest that future advancements in redundancy elimination will rely more on improving integration, adaptability, and cost awareness than on creating entirely new techniques. Recent research has begun to explore adaptive optimization strategies, iterative analyses with bounded cost, and compiler feedback mechanisms that dynamically assess the profitability of redundancy elimination (Liu et al., 2022; Wang et al., 2022).

Furthermore, as shown by Cai (2025), grammatical and structural decompositions of control-flow graphs offer promising avenues for exposing redundancy while limiting analytical complexity. These approaches point toward a future in which redundancy elimination is guided by richer program representations and informed by empirical performance data rather than static heuristics alone.

6. Conclusion

Redundancy elimination remains a fundamental yet evolving component of compiler optimization. While classical techniques such as CSE, LICM, PRE, and Value Numbering continue to form the foundation of optimization pipelines, this study shows that their efficacy is increasingly influenced by how they are integrated, ordered, and modified within modern compilation frameworks. Rather than treating redundancy elimination as a collection of isolated transformations, this research emphasizes its role as an interacting system of analyses whose influence depends on program structure, representation, and workload characteristics.

The primary contribution of this study lies in its analytical synthesis of redundancy elimination techniques in both foundational and modern contexts. Through the classification of redundancies, a unified framework for comparing elimination strategies, and an analysis of how these strategies interact with loop optimizations, the research demonstrates the practical limitations of known methodologies as well as their ongoing usefulness. In particular, the study also emphasizes that aggressive redundancy elimination is not inherently beneficial and may introduce inefficiencies when implemented without consideration for compilation costs or downstream optimization effects impacts.

One important finding from this work is that optimization profitability needs to be given top priority. Recent evidence shows that eliminating additional redundancies can degrade overall performance by increasing register pressure, interfering with vectorization, or complicating instruction scheduling. These call into question long-standing assumptions in compiler design and suggests a shift

toward cost-aware and context-sensitive optimization strategies. Therefore, redundancy elimination should be guided not only by correctness and availability but also by explicit models of benefit and interaction.

Looking forward, a number of promising avenues for further study emerge. First, especially in interactive and just-in-time compilation environments, adaptive redundancy elimination frameworks that dynamically balance optimization gain against analysis overhead are required. Secondly, opportunities to expose hidden redundancies while preserving analytical tractability are provided by richer program representations, such as SSA-based extensions, value graphs, and grammatical decompositions of control-flow graphs. Finally, domain-aware redundancy elimination techniques that go beyond traditional scalar and loop-centric assumptions are required for growing workloads in machine-generated code, data-intensive systems, and graph-based applications.

In conclusion, the compiler's capacity to integrate multiple techniques, analyze how they interact, and apply them selectively will determine how effective redundancy reduction remains. This work offers a perspective that is both practically grounded and forward-looking by defining redundancy elimination as an optimization problem defined by trade-offs rather than absolutes. It is hoped that the insights presented here will inform the design of more robust, efficient, and adaptable compiler optimization pipelines.

References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson Addison-Wesley.
2. Briggs, P., Cooper, K. D., Harvey, T. J., & Simpson, L. T. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8), 859–881.
3. Cai, X. (2025). Enhancing Compiler Optimization Efficiency through Grammatical Decompositions of Control-Flow Graphs. *arXiv preprint arXiv:2507.16660*.
4. Chow, F. (2021). Redundancy Elimination. In *SSA-based Compiler Design* (pp. 135-153). Cham: Springer International Publishing.
5. Cooper, K. D., & Torczon, L. (2012). *Engineering a compiler* (2nd ed.). Morgan Kaufmann.
6. Cooper, K. D., Eckhardt, J., & Kennedy, K. (2008). Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.
7. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 451–490. <https://doi.org/10.1145/115372.115320>
8. Dasgupta, D. M. (1991). Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2), 291–294.
9. Dasgupta, D. M. (2002). E-path-PRE: Partial redundancy elimination made easy. *SIGPLAN Notices*, 37(7), 53–65.
10. Dasgupta, S., & Gangwani, T. (2019). Partial redundancy elimination using lazy code motion. *arXiv*. <https://arxiv.org/abs/1905.08178>
11. Ding, Y., & Shen, X. (2017). GLORE: Generalized loop redundancy elimination upon LER-notation. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), Article 74. <https://doi.org/10.1145/3133898>
12. Enyindah, P., & Okon, E. U. (2017). The new trends in compiler analysis and optimizations. *International Journal of Computer Trends and Technology*, 46(2), 95.

13. Gao, F., Chen, H., Zhou, Y., & Wang, K. (2024, October). Shoot Yourself in the Foot—Efficient Code Causes Inefficiency in Compiler Optimizations. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1846-1857).
14. Jagdish, B. (2014). Effectiveness of inlining, common subexpression and dead code elimination in optimizing compilers. *International Journal of Computer Science and Network*, 3(1).
15. Jiang, S., Liang, Z., Sun, H., Shang, W., Tong, B., Yuan, M., ... & Li, Q. (2025). Lightweight Code Outlining for Android Applications. *ACM Transactions on Architecture and Code Optimization*.
16. Liu, H., Xu, J., Chen, S., & Guo, T. (2022). Compiler optimization parameter selection method based on ensemble learning. *Electronics*, 11(15), 2452. <https://doi.org/10.3390/electronics11152452>
17. Morel, E., & Renvoise, C. (1979). Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), 96–103. <https://doi.org/10.1145/359060.359070>
18. Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann.
19. Nabizath, S., & Vineeth, P. (2016). An improved algorithm for redundancy detection using global value numbering. *Journal of Information Processing Systems*, 12(2), 214–225. <https://doi.org/10.3745/JIPS.02.0014>
20. Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1988). Global value numbers and redundant computations. IBM Thomas J. Watson Research Center.
21. Roy, R., & Paleri, V. (2024). Partial Redundancy Elimination in Two Iterative Data Flow Analyses. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)* (pp. 35-1). Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
22. Van Drunen, T., & Hosking, A. L. (2004). Value-based partial redundancy elimination. In E. Duesterwald (Ed.), *Compiler Construction (LNCS Vol. 2985)*, pp. 167–184). Springer.
23. Wang, X., Wang, Y., Wan, Y., Mi, F., Li, Y., Zhou, P., Liu, J., Wu, H., Jiang, X., & Liu, Q. (2022). Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022* (pp. 9–19).
24. Yu, T. (2022). Unbalanced Big Data-Compatible Cloud Storage Method Based on Redundancy Elimination Technology. *Scientific Programming*, 2022(1), 1371778.
25. Yu, Z., Su, Z., Jiang, Y., Cui, A., & Wang, R. (2024, June). Efficient Code Generation for Data-Intensive Simulink Models via Redundancy Elimination. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (pp. 1-6).
26. Zhang, L., Jia, M., Gong, P., Lai, Z., Li, D., Fu, Y., Shen, A. & Lu, K. (2025). PDGNN: Efficient Micro-batch GNN Training via Degree-Pruned Partitioning and Redundancy Elimination. *ACM Transactions on Architecture and Code Optimization*, 22(4), 1-25.