ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25jun423

A Review of Binary Optimization in COBOL

Harshhitha Pattapuchetty[1]; Usha Rani K R[2]

[1]Student; [2]Professor
[1;2]Department of Electronics and Communication Engineering,
RV College of Engineering, Bengaluru, India

Publication Date: 2025/06/13

Abstract: Legacy software systems, specifically those that use COBOL, remain foundational in fields like banking, insurance, and government infrastructure. As hardware architectures evolve, optimizing legacy binaries while maintaining compatibility becomes increasingly important to harness the full potential of modern systems. However, direct refactoring of source code is often infeasible due to risk, cost, or unavailability of the original codebase. Binary optimization offers a compelling alternative that enables performance improvements at the binary level without touching the source. COBOL, one of the oldest high-level programming languages, continues to run critical workloads in sectors such as finance, government, and infrastructure. Despite its reliability, COBOL systems face growing challenges due to aging codebases, limited source code availability, and the increasing gap between legacy software and modern hardware capabilities. Recompilation is often infeasible, prompting the need for safe and practical binary-level optimization techniques. This paper presents a design-focused review of binary optimization in the context of COBOL, beginning with an overview of the language's execution model and architectural constraints. We discuss IBM's Automatic Binary Optimizer (ABO), a production-grade tool designed to enhance the performance of COBOL binaries on modern IBM Z systems without requiring source code. ABO's use of "smart binaries" and instruction-level metadata enables advanced validation workflows. We also survey key research contributions such as optimize-time validation (Koju et al.), hardware idiom recognition, adaptive runtime frameworks like COBRA, and formal verification tools like Alive2.

Keywords: *COBOL, Legacy Systems, Binary Optimization, IBM Automatic Binary Optimizer (ABO), Smart Binary, Optimize- Time Validation, Intermediate Representation, Mainframe Modernization, Packed Decimal Arithmetic, Runtime Validation, Static Analysis, Translation Validation, Dynamic Optimization.*

How to Cite: Harshhitha Pattapuchetty; Usha Rani K R (2025) A Review of Binary Optimization in COBOL. *International Journal of Innovative Science and Research Technology*, 10(6), 285-289.
https://doi.org/10.38124/ijisrt/25jun423

I. INTRODUCTION

COBOL (Common Business-Oriented Language) is one of the earliest high-level programming languages, developed in the late 1950s to support business and administrative systems. Despite its age, COBOL remains central to many industries, including banking, government, and insurance. Its persistence is largely due to its accuracy in fixed-point arithmetic, structured data handling, and its design for long term stability. One of the defining features of COBOL, its use of packed decimal (BCD) arithmetic, remains essential in applications where even small rounding errors are unacceptable. As shown in Figure 1, the evolution of binary optimization for legacy languages has accelerated in recent decades.

Most COBOL systems today are deployed on IBM main- frames running z/OS. While these systems continue to perform reliably, organizations face growing challenges: maintaining decades-old codebases, integrating with modern infrastructure, and improving execution speed on evolving hardware. In many cases, recompilation is not an option, due to lost source code, outdated compilers, or fears of breaking production systems. This has led to increased interest in techniques that can optimize COBOL applications at the binary level, preserving correctness while improving runtime performance.

Binary optimization refers to enhancing compiled executables without modifying the original source code. This approach is especially valuable for legacy systems, where source-level modification is risky or impossible. Binary optimizers work by disassembling machine code, reconstructing control flow, and applying transformations, either statically or dynamically, while preserving the program's behavior. However, achieving both safety and performance is a non-trivial task. Legacy binaries often contain undocumented behavior, complex idioms, or rely on hardware-specific quirks, making validation essential.

Among the most mature tools in this space is IBM's Automatic Binary Optimizer (ABO), which enhances COBOL binaries compiled by older Enterprise COBOL compilers. ABO restructures instruction sequences to make

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25jun423

use of modern IBM Z hardware features like vectorization, pipelining, and improved branch prediction without requiring access to source code. Crucially, ABO introduces the concept of a "smart binary," which includes metadata such as intermediate language mappings and I/O annotations. This metadata enables safe transformation and lays the groundwork for automated validation.

A key advancement on top of binary optimizers is Koju et al.'s optimize-time validation technique. Their method tests the compatibility by comparing the behavior of original and optimized binary fragments using multiple test inputs during compilation. If the outputs are equivalent, the optimized version is kept. Otherwise, the original is retained to preserve correctness. This allows developers to apply optimizations with strong behavioral guarantees, even in sensitive COBOL applications. Complementary to this, Alive2 by Lopes et al. introduces a formal, bounded translation validation system for LLVM that verifies the correctness of compiler optimizations using SMT solvers offering precision in a different, source-aware context. In the realm of test input generation, tools like Korat and CUTE enable automated validation through predicate-based and concolic testing, respectively. Meanwhile, dynamic optimizers such as Dynamo and COBRA implement runtime profiling and optimization, improving binaries adaptively without requiring static recompilation. Together, these works represent a rich and evolving landscape of solutions aimed at extending the life and performance of legacy applications, particularly those written in COBOL, through innovations in validation, analysis, and transformation of binaries. Together, these innovations highlight a convergence around key principles in binary optimization: the use of intermediate representations, validation-driven transformation, and hardware-aware enhancement.

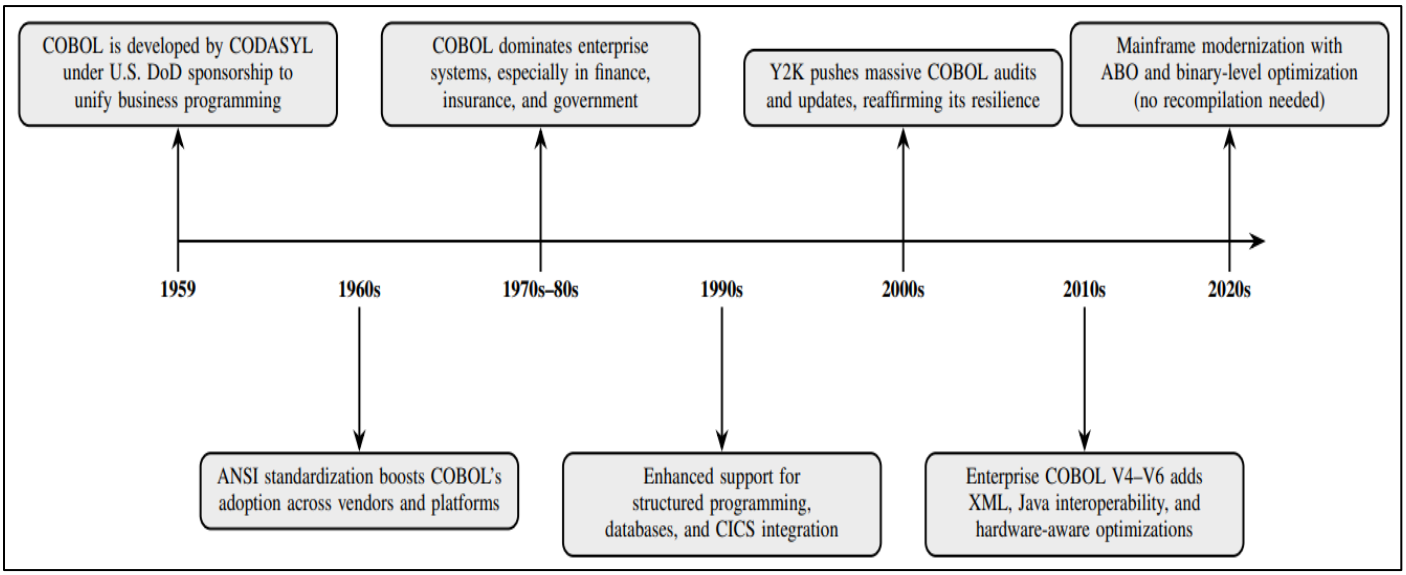

Fig 1 Timeline of Historical Developments in COBOL

II. BACKGROUND AND RESEARCH

➢ *COBOL and its Enduring Presence of Legacy Code*

COBOL, short for Common Business-Oriented Language, has stood the test of time in ways few programming languages have. Originally developed in the late 1950s, COBOL was designed with a specific vision in mind: to bridge the gap between computing systems and business professionals. Its English-like syntax made it easier for non-technical users to follow and even write code, which was revolutionary at the time [**?**]. Decades later, this characteristic has ironically become one of the reasons for its endurance. While newer languages have come and gone, COBOL remains a niche subject and deeply embedded in critical industries such as finance, banking, insurance, and government systems.

Today, COBOL systems still power almost 90% of global transactions, particularly through IBM Z platforms [**?**]. These programs often handle tasks like payroll processing, tax calculations, and ATM transactions—systems that are required to run with near-perfect reliability. Over time, these systems have grown into massive, interdependent codebases—sometimes with millions of lines of code, many of which are undocumented or poorly understood. The people who originally developed them have long since retired, and fewer new developers are trained in COBOL, creating a skills gap that makes modernizing these systems a major challenge.

Despite its age, COBOL is not 'obsolete' in the traditional sense. Rather, it has evolved alongside the hardware and enterprise environments it supports. Yet the reality remains that maintaining or upgrading COBOL systems is increasingly difficult. In many cases, the source code has been lost or locked behind outdated tooling, making even simple recompilation a high-risk operation. As a result, modern strategies for improving the performance or maintainability of these systems must often work with only the compiled binary, treating the executable itself as the starting point for optimization.

➢ *The Role of Binary Optimization*

This is where binary optimization comes into play. Binary optimization refers to the enhancement of compiled

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25jun423

binaries without recompilation, often by directly modifying executable code to improve runtime efficiency [**?**]. Instead of rewriting or recompiling a program, binary optimizers work directly on the machine code that runs on the hardware. This can include rewriting instruction sequences to be more efficient, making better use of hardware features like vector units or caches, or even reducing the overall size of the binary.

Binary optimization has gained renewed importance as legacy software systems continue to run critical workloads. In particular, when the source code is no longer available or when recompiling it introduces too much risk, binary optimization offers a practical alternative. Rather than rebuilding from scratch, binary optimization enhances what already works, extending the life of existing software while bringing it closer to modern performance expectations.

That said, optimizing binaries is far from trivial. Without the benefit of high-level context like variable names, data types, or control structures, the optimizer has to reverse-engineer the meaning of raw instructions [**?**]. Special care must be taken to preserve the program's original behavior, especially in fields like finance where even minor discrepancies in numeric precision can lead to major issues. Additionally, different architectures (such as IBM's z/Architecture used in mainframes) may introduce specialized instruction sets like those handling packed decimal operations in COBOL, which add further complexity [**?**].

- *IBM's Automatic Binary Optimizer (ABO)*

In response to these challenges, IBM developed the Automatic Binary Optimizer (ABO) to target performance bottle- necks in COBOL binaries compiled by older versions of its Enterprise COBOL compiler [**?**]. What makes ABO especially compelling is that it operates without needing the source code. Instead, it takes compiled load modules as input, analyzes them, and produces a semantically equivalent but performance- enhanced binary that is better suited for modern IBM Z hardware.

ABO works by recognizing common instruction pat-terns—particularly those emitted by legacy compilers—and replacing them with more efficient equivalents. These trans-formations exploit advances in microarchitectural features such as instruction pipelining, branch prediction, and decimal arithmetic accelerators [**?**]. Since many COBOL programs were compiled years or even decades ago, they often contain instruction sequences that are suboptimal for today's hardware. ABO addresses this gap without altering the application's logic or requiring redevelopment.

One of the key features of ABO is its focus on safety and compatibility. Before applying any transformation, the optimizer performs a validation step, which compares the execution results of the original and optimized code fragments with several input data points [**?**]. If the results differ, the optimizer falls back to the original code to ensure correctness. To support deeper optimization and validation, ABO utilizes the concept of a *smart binary*—a binary enriched with metadata about intermediate language (IL) mappings and I/O descriptors. These mappings are crucial for validation because they help the optimizer reason about what registers or memory areas a code snippet uses or modifies [**?**]. This additional information enables optimize-time validation, which ensures that only verified code substitutions are retained in the final output.

In short, ABO is a practical and well-engineered solution for organizations that rely on COBOL systems but want to benefit from hardware-level performance improvements. It serves as a bridge between the legacy world of COBOL and the performance demands of modern enterprise computing.

III. ADVANCEMENTS IN COBOL AND BINARY OPTIMIZATION: A REVIEW OF RECENT INNOVATIONS

COBOL continues to power a vast portion of the digital infrastructure behind modern commerce, finance, and govern- ment. Its legacy, however, is increasingly at odds with the pace of hardware innovation and the expectations of modern software tooling. While rewriting COBOL applications in modern languages is one approach, it is often impractical due to cost, risk, and the sheer volume of code. As such, binary optimization has emerged as a practical and low-risk strategy to breathe new life into aging systems.

Over the last two decades, researchers and engineers have proposed increasingly refined methods to improve binaries post-compilation, particularly for COBOL programs running on IBM mainframes. This section surveys five key works that have shaped the current thinking around optimizing and validating COBOL binaries. These include execution-based validation during optimization, hardware idiom recognition, dynamic profiling systems, language-specific binary annotation, and formal translation validation frameworks.

- *Automatic Optimize-Time Validation for Binary Optimizers (Koju et al., 2020)*

Koju et al. propose a technique called *optimize-time validation* to ensure that optimized binary segments preserve behavior identical to the original code [**?**]. Their strategy is unique in that it validates transformations during the compilation process itself, executing both versions with different inputs and comparing their outputs.

The optimizer selects small code regions—between 1 and 100 instructions—and generates both optimized and fallback paths. It then performs inline validation to decide which path to include in the final binary. This runtime comparison enables safe and automated transformation while retaining correctness. Supporting this is the concept of a *smart binary*, which embeds metadata such as IL mappings and memory/register usage into the binary itself. This context allows the optimizer to reason more precisely about the effects of code transformations [**?**].

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25jun423

The authors demonstrated that their method could catch previously unknown bugs and introduce minimal performance overhead—just 0.02% in compile time and an 8% binary size increase—while guaranteeing behavioral compatibility.

- *A New Idiom Recognition Framework for Hardware Optimization (Kawahito et al., 2006)*

To bridge the gap between legacy code and modern hard- ware features, Kawahito et al. developed an idiom recognition framework that identifies patterns in machine code and maps them to hardware-assisted instructions [**?**]. Idioms like multi- instruction decimal formatting loops can be replaced by efficient hardware operations such as the IBM Z ED instruction. The system uses a rule-based engine to scan for recognizable sequences and then rewrites them using optimized templates. Because COBOL often compiles to repetitive arithmetic and formatting instructions, this approach is particularly effective in financial domains.

Experimental results show substantial reduction in instruction count and runtime latency for idiomatic code, affirming the viability of idiom-driven optimization.

- *COBRA: Runtime Adaptive Optimization Framework (Kim et al., 2007)*

COBRA takes a dynamic approach, introducing optimization during program execution instead of at compile time. By monitoring hot paths at runtime, the system performs just-in- time optimizations on frequently executed code segments [**?**].

The framework utilizes profile-guided techniques to detect performance bottlenecks and applies transformations such as inlining and loop unrolling while the application is live. This is especially useful for long-running COBOL batch processes where predictable execution patterns emerge over time.

Their benchmark tests reported speedups ranging from 10–30% with minimal runtime overhead. This suggests a promising use case for adaptive optimization in enterprise systems, even when recompilation is not feasible.

- *Re-Constructing High-Level Information from Binaries (Koju et al., 2016)*

This earlier work by Koju et al. lays the foundation for smart binary optimization by embedding high-level information in binaries at compile time. It enables tools to "lift" machine code into semantically meaningful forms [**?**].

They propose that compilers emit mappings from IL nodes to assembly instructions and record which memory areas or registers each node uses. This enriched binary structure, termed a *language-specific binary*, is essential for applying source-aware transformations post-compilation.

By maintaining the relationships between original IL and low-level code, optimizers like ABO can validate transformations more effectively and apply aggressive but safe enhancements.

- *Alive2: Bounded Translation Validation for Compiler Optimizations (Lopes et al., 2021)*

Alive2 is a tool that performs formal translation validation by comparing the logical equivalence of original and optimized code. While designed for LLVM, the technique provides a useful model for verifying binary transformations [**?**].

The system encodes source and target programs as logical constraints and uses SMT solvers to prove that their behaviors are identical. This method captures subtle correctness issues that traditional testing might miss.

Although not yet adapted for COBOL or binary code directly, Alive2 demonstrates how formal verification tools can enhance trust in compiler and optimizer output, and could inform future developments in COBOL binary analysis frameworks.

IV. CONCLUSION

Over the decades, COBOL has cemented its place as a core component of enterprise computing. However, as the demands of modern hardware and systems have evolved, the limitations of maintaining and optimizing such legacy systems have become more pronounced. In this review, we have traced the landscape of COBOL binary optimization, beginning with the language's architecture and its constraints, then moving through practical techniques like IBM's Automatic Binary Optimizer, and finally exploring cutting-edge academic contributions that address validation, idiom recognition, and formal correctness.

What emerges is a picture of steady, meaningful progress: researchers have developed both practical tools and theoretical frameworks to safely modernize binary code without source access. Techniques like optimize-time validation bridge the gap between engineering pragmatism and correctness assurance, while frameworks like COBRA and Alive2 expand the horizon of what's possible in binary transformation.

This review not only highlights the strengths and limitations of existing approaches but also provides a consolidated foundation for further work in the field. Whether one approaches COBOL from a performance, reliability, or educational perspective, it is clear that legacy software is not going away and the research community must continue to build smarter, safer tools to support it.

FUTURE SCOPE

Despite significant progress, binary optimization for legacy languages like COBOL remains a field full of open questions and technical hurdles.

One of the most immediate challenges is the limited access to real-world COBOL binaries and execution environments. Tools like ABO are proprietary and run only on IBM z/OS systems, making it difficult for researchers and students to experiment, test, or extend existing ideas.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25jun423

Additionally, the lack of standard benchmarks and open datasets for COBOL binaries limits reproducibility and evaluation of optimization techniques.

Another major challenge lies in the precision of validation, particularly for COBOL's decimal arithmetic operations. Packed decimal instructions, conditional edits, and implicit rounding rules introduce edge cases that are difficult to model and test exhaustively. While black-box and specification-based testing approaches have proven practical, formal verification at the binary level especially for non-trivial programs, remains largely unsolved. Future work could include developing open- source alternatives to smart binaries, lifting COBOL binaries into generic IRs for symbolic reasoning, or hybrid tools that combine static analysis with runtime profiling. There is also a strong case for making binary optimization more accessible to the education and research communities by creating simplified simulation environments or interactive visualizations of binary transformations.

Finally, as more legacy systems become candidates for containerization or cloud migration, there is potential to combine binary optimization with modern deployment strategies, enabling old programs to run efficiently in new contexts without recompilation. These intersections of systems engineering, programming languages, and formal methods will define the next wave of research in COBOL optimization and legacy software transformation.

REFERENCES

[1]. T. Koju, R. Copeland, M. Kawahito and M. Ohara, "Automatic Optimize-Time Validation for Binary Optimizers," *Proceedings of the 2020 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Diego, CA, USA, 2020, pp. 137–148, doi: 10.1145/3368826.3377919.

[2]. M. Kawahito, H. Komatsu, T. Moriyama, H. Inoue and T. Nakatani, "A New Idiom Recognition Framework for Exploiting Hardware- Assist Instructions," *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, USA, 2006, pp. 382–393, doi: 10.1145/1168857.1168902.

[3]. J. Kim, W. Hsu and P. Yew, "COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications," *Proceedings of the 2007 International Conference on Parallel Processing (ICPP)*, Xi'an, China, 2007, pp. 25–32, doi: 10.1109/ICPP.2007.23.

[4]. T. Koju, R. Copeland, M. Kawahito and M. Ohara, "Re- constructing High-Level Information for Language-Specific Binary Re-Optimization," *Proceedings of the 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Barcelona, Spain, 2016, pp. 273–283, doi: 10.1145/2854038.2854057.

[5]. N. Lopes, J. Lee, C.-K. Hur, Z. Liu and J. Regehr, "Alive2: Bounded Translation Validation for LLVM," *Proceedings of the ACM on Program- ming Languages (POPL)*, Vol. 5, 2021, pp. 1–29, doi: 10.1145/3434304.

[6]. J. E. Sammet, "Brief Summary of the Early History of COBOL," in Annals of the History of Computing, vol. 7, no. 4, pp. 288-303, Oct.- Dec. 1985, doi: 10.1109/MAHC.1985.10044.

[7]. IBM Corporation, "Migrating to Enterprise COBOL V6," IBM Documentation, 2017. [Online]. Available: https://www.ibm.com/docs/en/cobol-zos/6.3

[8]. IBM Corporation, "z/Architecture Principles of Operation," IBM Redbooks, SA22-7832-12, 2017. [Online]. Available: https://www.ibm.com/docs/en/zos/2.3.0?topic=architecture- zarchitecture-principles-operation

[9]. S. S. Muchnick, *Advanced Compiler Design and Implementation*, San Francisco, CA, USA: Morgan Kaufmann, 1997.

[10]. V. Bala, E. Duesterwald and S. Banerjia, "Dynamo: A Transpar- ent Dynamic Optimization System," *Proceedings of the ACM SIG- PLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, 2000, pp. 1–12, doi: 10.1145/349299.349303.

[11]. D. Bruening, T. Garnett and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," *International Symposium on Code Generation and Optimization (CGO)*, 2003, pp. 265–275, doi: 10.1109/CGO.2003.1191553.

[12]. K. Sen, D. Marinov and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proceedings of the 10th European Software Engineering Conference*, 2005, pp. 263–272, doi: 10.1145/1081706.1081750.

[13]. C. Boyapati, S. Khurshid and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, 2002, pp. 123–133, doi: 10.1145/566172.566191.

[14]. J. C. King, "Symbolic Execution and Program Testing," *Commu- nications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976, doi: 10.1145/360248.360252.

[15]. LLVM Project, "The LLVM Compiler Infrastructure." [Online]. Avail- able: https://llvm.org/

[16]. M. J. Gordon and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.