Optimization of Short-Circuit Evaluation for High-Performance and Energy-Efficient Computing

[1] Adewole A. Philip, [2]Durojaye E. Olatunji, [3]Agoreyo M. Neekay, [4]Oluwawibe C. Olatomiwa, [5]Owoyomi O. Emmanuel

[1]Corresponding Author, [2]First-author, [3]Co-author, [4]Co-author, [5]Co-author
[1]Department of Computer Science,
[1]University of Lagos, Akoka, Nigeria
[1]padewole@unilag.edu.ng , [2]durojayeemmanuelolatunji@gmail.com, [3]marehscepe@gmail.com, [4]tomiwa.oluwawibe@gmail.com , [5]owoyomioluwasegun0@gmail.com

***Abstract*— Short-circuit evaluation is a widely used programming-language mechanism that avoids unnecessary computation during Boolean expression evaluation. This article synthesizes a structured review of thirty key works in compiler optimization, lazy evaluation, Boolean logic execution, machine-learning-driven compiler techniques, and energy-efficient computing to develop optimization strategies for short-circuit evaluation. The paper identifies empirical and methodological gaps in existing literature, proposes a conceptual optimization framework that combines static compiler passes with runtime adaptive ordering, and outlines an experimental agenda for validating performance and energy benefits in modern computing environments. The proposed framework demonstrates that optimized short-circuit evaluation can reduce computational overhead, improve execution efficiency, and contribute to energy-aware software engineering practices.**

***Index Terms*— Short-circuit, High-performance, Optimization, Computing, Programming, Efficiency.**

I. INTRODUCTION

Short-circuit evaluation is a fundamental optimization technique in programming languages that prevents unnecessary computation during Boolean expression evaluation. Short-circuit evaluation halts the evaluation of a Boolean expression as soon as the overall truth value is determined, thereby avoiding the execution of remaining subexpressions. In practice, this behavior reduces CPU cycles, memory accesses, and instruction execution, and it has implications for energy consumption in large-scale and high-performance systems [1] [14]. Despite its ubiquity, the literature has emphasized theoretical correctness, compiler design, and verification rather than empirical performance measurement and energy analysis. This article consolidates theoretical foundations and recent advances in compiler and machine-learning optimization to present a coherent framework for optimizing short-circuit evaluation and to propose a research program for empirical validation.

II. BACKGROUND AND RELATED WORK

Short-circuit evaluation rests on Boolean algebra and compiler optimization theory. Foundational compiler texts describe Boolean expression handling during intermediate-code generation and optimization [1]. Formal treatments of short-circuit logic have clarified non-commutative and evaluation-order semantics [3], while graph-based methods have been used to manipulate and optimize Boolean functions [7]. Research on lazy evaluation and execution skipping demonstrates that deferring or avoiding computations can yield substantial efficiency gains in functional languages and database query engines [12] [22]. Compiler research has shown that combining multiple optimizations passes and leveraging modern infrastructures such as MLIR enables more sophisticated transformations [8] [21]. Recent work applying machine learning to compiler heuristics and phase ordering suggests that adaptive, data-driven strategies can further improve optimization outcomes [29][13][15]. Architecture studies on branch prediction and basic block reordering provide additional evidence that evaluation ordering and execution-path control materially affect runtime performance [14][25]. Energy-efficiency research indicates that computation skipping reduces energy use, but direct empirical studies focused on short-circuit evaluation are scarce [23].

III. CONCEPTUAL FRAMEWORK AND METHODOLOGY

This work synthesizes the literature into a conceptual optimization framework that integrates static compiler analysis, cost modeling, evaluation ordering, and runtime adaptation. The framework comprises five stages: Boolean analysis to identify candidate expressions; cost analysis to estimate execution and memory costs and to model evaluation probabilities; static evaluation ordering to prefer low-cost, high-discriminative subexpressions; compiler-level transformations (dead-code elimination, constant folding, branch optimization); and runtime adaptive optimization that reorders evaluation based on observed execution frequencies.

Methodologically, the framework is grounded in comparative analysis of thirty representative works spanning compiler theory, architecture, lazy evaluation, and energy studies. Evaluation criteria include instruction count reduction, execution time, memory access patterns, branch behavior, and energy consumption. The proposed experimental agenda calls for microbenchmarks and real-world program suites, energy measurement using platform power counters, and controlled experiments across CPU, cloud, and GPU environments to quantify trade-offs and generalizability.

IV. ANALYSIS AND DISCUSSION

Optimizing evaluation order can yield measurable reductions in instruction count and execution time by ensuring that inexpensive or highly discriminative checks are evaluated first, thereby increasing the probability that later, more expensive checks are skipped. Compiler passes that expose and exploit short-circuit opportunities - combined with profile-guided or machine-learned ordering - can amplify these gains. Runtime adaptive mechanisms that monitor branch outcomes and recompile or reorder hot expressions can respond to changing input distributions and workload characteristics, improving long-running application efficiency.

Energy benefits follow from reduced instruction execution and lower CPU utilization. While prior work on computation skipping supports this claim [23], the literature lacks direct, systematic measurements tying short-circuit optimization to energy savings across representative workloads. Integrating energy measurement into benchmarking pipelines is therefore essential. Opportunities for future systems include compiler plug-ins that implement cost-aware reordering, reinforcement-learning agents that learn ordering policies from runtime traces, and cloud-native optimizations that exploit workload-level statistics to tune evaluation strategies.

Limitations of current research include a predominance of theoretical treatments, a shortage of modern benchmarks that reflect contemporary language features and runtime systems, and limited cross-platform validation. Addressing these limitations requires a coordinated empirical program combining compiler engineering, instrumentation for energy measurement, and machine-learning experimentation.

V. CONCLUSION AND RECOMMENDATIONS

Short-circuit evaluation is an underexploited lever for improving both performance and energy efficiency in modern computing systems. This article presents a conceptual framework that unites static compiler techniques, cost modeling, and runtime adaptation, and it articulates an empirical agenda to validate and refine these ideas. To advance the field, researchers should prioritize the development of benchmark suites that include realistic Boolean-heavy workloads, integrate precise energy measurement into evaluation pipelines, and explore AI-driven ordering strategies that generalize across programs and platforms. Practical next steps include implementing prototype compiler passes that perform cost-aware reordering, instrumenting runtimes to collect evaluation statistics, and conducting controlled experiments on CPU, cloud, and GPU targets.

REFERENCES

[1] Aho, A. V., Lam, M., Sethi, R., & Ullman, J. (2006). *Compilers: Principles, techniques and tools*. Pearson.

[2] Allen, R., & Kennedy, K. (2002). *Optimizing compilers for modern architectures*. Elsevier.

[3] Bergstra, J. A., Ponse, A., & Staudt, D. (2021). Non-commutative propositional logic with short-circuit evaluation. *Journal of Applied Non-Classical Logics*.

[4] Bieri, A., Cimatti, A., Clarke, E., & Zhu, Y. (2003). Symbolic model checking without BDDs. In *Proceedings* (Springer).

[5] Bodik, R., Gupta, R., & Soffa, M. (2000). Load reuse analysis. *ACM*.

[6] Burch, J., & Dill, D. (2000). Automatic verification of pipelined microprocessor control. *ACM*.

[7] Bryant, R. (2001). Graph-based algorithms for Boolean function manipulation. *IEEE*.

[8] Click, C., Cooper, K., & Kennedy, K. (2000). Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*.

[9] Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ANSI-C programs. In *Proceedings* (Springer).

[10] Cooper, K., & Torczon, L. (2012). *Engineering a compiler*. Elsevier.

[11] Cousot, P., & Cousot, R. (2002). Abstract interpretation. *ACM*.

[12] Garcia, R., Lumsdaine, A., & Sabry, A. (2010). Lazy evaluation and delimited control. *arXiv*.

[13] Haj-Ali, A., Huang, Q., Moses, W., Stoica, I., & Asanovic, K. (2019). AutoPhase: Compiler phase ordering for high level synthesis with deep reinforcement learning. *arXiv*.

[14] Hennessy, J., & Patterson, D. (2019). *Computer architecture: A quantitative approach*. Elsevier.

[15] Huang, Q., Haj-Ali, A., Moses, W., Stoica, I., & Asanovic, K. (2020). AutoPhase: Juggling HLS phase ordering. *arXiv*.

[16] Homolya, M., Mitchell, L., Luporini, F., & Ham, D. (2017). TSFC: A structure-preserving form compiler. *arXiv*.

[17] Jones, S. P., Hughes, J., et al. (2003). *Haskell 98 language and libraries*. Cambridge University Press.

[18] Kroening, D., & Strichman, O. (2016). *Decision procedures: An algorithmic point of view*. Springer.

[19] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Walter, K., & Bala, K. (2008). Optimistic parallelism. *ACM*.

[20] Kumar, S., & Pande, S. (2000). Compiler optimizations for improving data locality. *ACM*.

[21] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., et al. (2020). MLIR: A compiler infrastructure for the end of Moore's law. *arXiv*.

[22] Maurer, D. (2007). Lazy execution of Boolean queries. In *Lecture Notes in Computer Science* (Springer).

[23] Mittal, S. (2014). A survey of techniques for improving energy efficiency in computing systems. *arXiv*.

[24] Moseley, T., Marks, J., Wonnacott, D., & Anderson, S. (2006). LoopProf: Dynamic techniques for loop optimization. *ACM*.

[25] Newell, A., & Pupyrev, S. (2018). Improved basic block reordering. *arXiv*.

[26] Ottoni, G. (2005). Global value numbering. *LLVM Project*.

[27] Patterson, D., & Hennessy, J. (2013). *Computer organization and design*. Elsevier.

[29] Stephenson, M., Amarasinghe, S., Martin, M., & O'Reilly, U. (2005). Meta optimization: Improving compiler heuristics. *ACM*.

[30] Vechev, M., Yahav, E., & Yorsh, G. (2010). Experience with model checking optimized code. *ACM SIGPLAN*.

[31] Wilhelm, M. (2017). Balancing expression DAGs for more efficient lazy adaptive evaluation. *arXiv*.