iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.81428

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

LOCKE - Framework for Bit-Level Logic Optimization, Compilation and Execution

Gowri Sankar A[1], Dr. Thirumahal R[2], Naveen Kumar A[3], Vanjinathan P[4], Sibi M[5], Gowtham P[6]
Department of Computer Science and Engineering, PSG College of Technology, Coimbatore, India

Abstract: ***The proposed framework, LOCKE (Logic Optimization, Compilation, and Execution), enables efficient expression and execution of bit-level computation by transforming programs into a fully dependency-resolved Boolean representation. Instead of relying on control flow and sequential execution, the system models computation as a flattened graph of logic operations, exposing complete dependencies and eliminating ambiguity. The framework follows a structured pipeline consisting of parsing, logic lowering, static scheduling, and code generation, ensuring that execution order is resolved entirely at compile time. This approach allows deterministic execution, improved parallelism, and reduced runtime overhead. By shifting computation from a control-driven model to a logic-driven paradigm, LOCKE achieves higher efficiency and scalability for performance-critical bit-level applications.***
Keywords: bit-level computation, logic optimization, dependency graph, static scheduling, deterministic execution, compiler design

I. INTRODUCTION

As computing systems become more sophisticated, they are increasingly based on low-level data manipulation. In application areas like cryptography, networking, embedded systems, and high-performance computing, computation is not only defined on values but also on discrete bits that represent structure and control information. Be it assembling communication protocol headers, manipulating encryption data, or manipulating hardware signals, bit-level correctness and efficiency are essential. In these domains, computing is inherently fine-grained and even minor inefficiencies can snowball into performance problems.

Yet, bit computation is relegated to a lower standing in most programming environments. Programmers must construct bit operations by using low-level bit masks, bit shifts and subscripts within word-sized abstractions. This representation, while general, is opaque and makes it hard to understand, debug and optimize programs. Compilers, in particular, do not explicitly expose bit-level dependency information, especially for the purposes of more aggressive optimizations such as Boolean simplification and parallel execution.

Existing compilers are built around the notions of control flow, mutable state and sequential execution. Software is modelled as streams of instructions where order

of execution is dynamically determined, often leading to spurious dependencies and limiting global optimisations. While this approach works well for general-purpose computation, it struggles for logic-intensive applications, where computation is driven by data dependencies rather than control.

A. Objective of the Proposed Format

The shortcomings of current solutions suggest that bit-level computation is missing a well-organised and expressive model that captures intent and supports efficient execution. In existing systems, simple operations must be broken down into multiple instructions, which results in redundant computation, a large number of instructions and poor parallelism. Also, the order of execution is often dynamic, causing overhead and unpredictable execution.

In many real-world scenarios, such inefficiencies become critical. For example, cryptographic algorithms involve repeated bitwise transformations where redundant operations can significantly impact throughput. Networking systems require precise and fast manipulation of packet headers, while embedded systems operate under strict resource constraints where both memory and power efficiency are crucial. In all these cases, the inability to represent and optimize computation at the bit level leads to suboptimal performance.

B. An Overview of Existing Formats

Researchers and practitioners have explored various methods to improve low-level computation, primarily through compiler optimizations and intermediate representations. Techniques such as Static Single Assignment (SSA), graph-based intermediate representations, and instruction-level parallelism aim to expose dependencies and enable optimization. While these approaches improve certain aspects of execution, they still operate within the constraints of traditional instruction-based models.

Even advanced optimizations are typically applied after instruction generation, limiting their ability to perform global transformations. Control flow remains embedded in the representation, and mutable variables introduce complexities in dependency analysis and scheduling. As a result, many optimization opportunities at the logic level remain unexplored.

Another line of work draws inspiration from hardware design, where computation is represented as dataflow graphs or circuits. These models naturally expose dependencies and enable parallel execution. However, they are often restricted to specialized tools or hardware synthesis environments and are not easily integrated into general-purpose programming workflows.

C. The Scope of the Proposed Format

The gap between software-centric and hardware-centric approaches motivates the need for a unified framework that combines expressiveness with efficiency. In this paper, we present **LOCKE (Logic Optimization, Compilation, and Execution)**, a framework designed to transform bit-level programs into a deterministic, dependency-driven execution model.

Rather than representing programs as sequences of instructions, LOCKE models computation as a network of Boolean operations, where each operation is defined by its inputs and outputs. The system introduces a domain-specific language that allows direct and unambiguous expression of bit-level operations such as slicing, concatenation, and indexing. This representation preserves semantic clarity while eliminating unnecessary abstraction.The framework follows a structured pipeline consisting of four stages. The first stage constructs the program structure, followed by a transformation into a flattened Boolean dependency graph that captures all computational relationships. This graph undergoes optimization to eliminate redundancy and reduce complexity. The optimized representation is then statically scheduled based on dependency analysis, organizing operations into execution stages that can be executed in parallel. Finally, the system generates efficient executable code using a register-based model.

This approach leads to several key advantages. By eliminating control flow and resolving dependencies at compile time, execution becomes deterministic and predictable. The explicit dependency graph enables fine-grained parallelism, allowing both data-level and thread-level execution. Resource utilization is improved through efficient register reuse, and overall computational overhead is reduced by avoiding redundant operations.

D. Background and Related Work

The challenges addressed by this work stem from a mismatch between how computation is expressed and how it is executed. Traditional programming models emphasize control flow and sequential execution, whereas many real-world workloads are inherently parallel and dependency-driven. This mismatch results in inefficiencies that become more pronounced in performance-critical applications. By shifting the focus from control-driven execution to logic-driven evaluation, LOCKE bridges the gap between software and hardware paradigms. Computation is no longer viewed as a sequence of instructions but as a structured network of dependencies that can be analyzed, optimized, and executed efficiently. This perspective enables a new class of optimizations that are not possible in conventional compiler pipelines.

The remainder of this paper presents the design and implementation of the LOCKE framework. We begin by examining the limitations of existing approaches and motivating the need for a logic-centric model. We then describe the language design, compilation stages, and optimization techniques in detail. Finally, we evaluate the system using representative workloads, demonstrating improvements in efficiency, scalability, and execution performance.

II. LITERATURE REVIEW

A. Literature Survey of Research Papers

This paper investigates the use of compiler intermediate representations (IRs) as a foundation for deep program analysis and transformation, particularly in the context of optimization-driven learning systems. The authors argue that source-level representations contain excessive syntactic noise, which hinders accurate semantic reasoning.

By converting programs into optimized LLVM intermediate representation, the compiler exposes control-flow normalization, explicit data dependencies, and canonicalized computation graphs. A genetic algorithm is employed to determine optimal sequences of compiler passes that enhance semantic stability and structural consistency of the IR. The study demonstrates that normalized IRs significantly improve downstream tasks such as similarity detection and program classification. The work implicitly reinforces a critical compiler design principle: semantic flattening and representation uniformity are essential for effective optimization. Nevertheless, its emphasis on IR-level determinism directly supports logic-centric compilation pipelines where computation is expressed as dependency graphs rather than control-flow constructs. [1]

This survey presents a detailed taxonomy of intermediate representations designed for heterogeneous and multicore computing systems. The authors classify IRs into linear, tree-based, and graph-based forms, evaluating their suitability for optimization, parallelism extraction, and architectural mapping. Special emphasis is placed on graph-based IRs due to their explicit encoding of data dependencies and reduced reliance on control flow. The paper highlights that dependency graphs enable more aggressive transformations such as global scheduling, speculative execution, and parallel mapping. Notably, the paper identifies that flat, acyclic representations are particularly effective for deterministic execution environments. While hardware execution is discussed only abstractly, the work provides strong justification for compiler pipelines that eliminate high-level abstractions early and operate on dependency-resolved graphs. [2]

This paper focuses on instruction-level parallelism (ILP) extraction techniques for superscalar and pipelined processors. The authors analyze dependence analysis, loop unrolling, software pipelining, and instruction scheduling as primary mechanisms for maximizing throughput. Experimental results demonstrate measurable performance gains on pipelined architectures when scheduling is applied aggressively. However, the study assumes a conventional compiler pipeline where instructions are generated prior to scheduling, limiting global visibility. Control-flow constructs remain intact throughout the process. While effective for general-purpose processors, the approach does not scale naturally to logic-level representations where control flow is eliminated entirely. Nonetheless, the dependency-driven scheduling concepts directly inform static scheduling strategies used in virtual hardware execution models. [3]

This work addresses the interaction between register allocation and instruction scheduling, arguing that treating them as independent phases leads to suboptimal results. The authors propose a cooperative framework where scheduling decisions influence register pressure estimation and vice versa. By integrating feedback loops, the compiler minimizes spill costs and improves execution efficiency, particularly in embedded systems with limited registers. The study focuses on loop-intensive workloads and demonstrates reduced schedule length and memory traffic. Although the approach remains grounded in control-flow graphs and variable-based representations, it provides an important insight: resource allocation and scheduling must be considered together at a global level. The paper does not remove variables or procedural abstractions, nor does it flatten computation into logic graphs. However, its integrated philosophy aligns strongly with compiler pipelines that perform scheduling and register reuse directly
on dependency-resolved graphs. [4]

This paper evaluates the effectiveness of static instruction scheduling and register allocation strategies on
out-of-order processors. The authors show that aggressive compile-time scheduling often yields diminishing returns due to dynamic hardware scheduling mechanisms. Additionally, increased register pressure can negate performance gains through spill overhead. The study provides extensive empirical analysis across multiple benchmarks and architectures. Importantly, it highlights that compile-time determinism is less critical when sophisticated hardware support exists. However, the paper implicitly reveals the opposite case: in environments without dynamic scheduling or large register files, static compilation becomes essential. While the authors do not explore such environments explicitly, their results motivate compiler designs that assume minimal hardware intelligence. This
observation supports logic-level compilation pipelines where execution order, resource usage, and storage reuse are resolved entirely at compile time. [5]

This paper presents an SSA-based framework for pointer and alias optimization aimed at improving compile-time analysis precision. By transforming programs into Static Single Assignment form, the compiler enforces a single-definition property for variables, significantly simplifying data-flow analysis. The authors apply constant propagation, alias class formation, and redundancy elimination to reduce ambiguity in memory access patterns. Experimental results show that SSA-based optimizations improve both compilation speed and optimization effectiveness. Although the work remains within the traditional variable-based compilation paradigm, it demonstrates the power of enforcing immutability and explicit dependency relationships. However, its core principle—each value being defined exactly once—maps naturally to logic-graph representations where nodes represent immutable computations. The work strongly motivates the elimination of mutable state early in the compilation process, which is essential for deterministic scheduling and flattening. [6]

This study investigates graph reachability optimization techniques for large-scale static program analysis. The authors propose methods such as strongly connected component condensation, transitive edge reduction, and topological ordering to reduce analysis complexity. By simplifying dependency graphs, the approach significantly accelerates reachability queries and reduces memory overhead. While the paper is framed around source-code analysis and verification, its techniques are directly applicable to compiler optimization pipelines
operating on dependency graphs. However, it provides critical insights into maintaining scalability when dealing with large graphs. For compilers that flatten computation into logic DAGs, efficient graph simplification is essential to reduce scheduling complexity and improve register reuse decisions. The paper's contribution lies in its demonstration that structural graph optimization can be performed without altering program semantics. [7]

This paper explores interprocedural optimization through semantic inlining and function summarization. The authors propose replacing function calls with abstract summaries that capture essential behavior, enabling cross-procedural optimization without excessive code expansion. This approach improves constant propagation, dead-code elimination, and redundancy removal across function boundaries. While effective, the framework relies heavily on accurate semantic annotations and retains procedural abstraction. The work does not eliminate control flow or convert programs into flat computational graphs.
Nevertheless, it highlights the benefits of collapsing abstraction boundaries early to expose global optimization
opportunities. In compiler pipelines that completely remove procedural constructs, this work can be seen as an intermediate step. It reinforces the idea that abstraction elimination is necessary to achieve whole-program
optimization. [8]

This paper surveys adaptive query optimization techniques that dynamically adjust execution plans based on runtime feedback. The authors classify approaches into parametric, progressive, and adaptive models, each balancing flexibility and overhead. While effective in data-intensive systems, these techniques introduce runtime nondeterminism and monitoring overhead. The paper emphasizes that adaptability is valuable when data characteristics are unpredictable. However, it implicitly highlights the trade-off between flexibility and determinism. For systems requiring predictable execution, such as logic-level compilers and virtual hardware pipelines, runtime adaptation is undesirable. This work therefore motivates static resolution of execution structure at compile time. Although focused on databases, the principles generalize to compiler design decisions. [9]

This work presents compile-time optimization of object-oriented queries using metadata-driven heuristics. By estimating selectivity and access patterns, the compiler generates optimized execution plans ahead of runtime. The approach reduces query latency and improves resource utilization. However, the reliance on probabilistic metadata introduces uncertainty and limits determinism. The paper demonstrates that compile-time planning is beneficial when sufficient information is available. In contrast, logic-centric compilers assume full structural knowledge and avoid probabilistic decision-making. While the domain differs, the paper reinforces the value of early decision resolution to minimize runtime overhead. It provides indirect support for compilation strategies that fully determine execution behavior before runtime. [10]

This paper investigates metadata-driven optimization strategies that combine compile-time planning with limited runtime validation. The authors propose embedding metadata descriptors into intermediate representations to guide optimization decisions such as reordering, caching, and specialization. By incorporating feedback mechanisms, the compiler adapts execution strategies when metadata assumptions deviate from runtime behavior. The work demonstrates performance improvements in dynamic workloads but at the cost of increased system complexity. While effective in flexible execution environments, this hybrid model contrasts with fully static compilation approaches. The paper implicitly highlights that increased adaptability often conflicts with predictability. For compiler pipelines targeting deterministic execution and hardware-like semantics, eliminating runtime dependence is preferred. Nevertheless, the work contributes valuable insight into how compile-time representations can encode rich semantic information, reinforcing the importance of expressive IRs even in static systems. [11]

This paper proposes cache-aware compile-time optimization techniques that introduce result caching into program execution paths. The authors analyze computation reuse patterns and automatically insert cache structures to avoid redundant evaluations. Experimental results show significant performance improvements in repeated-query workloads. However, the approach introduces stateful behavior and relies on cache coherence policies. The compiler must manage cache invalidation and consistency, increasing complexity. While the work focuses on software systems, it demonstrates how computation reuse can be exploited when dependencies are explicitly modeled. The paper reinforces the importance of identifying common subexpressions and exploiting immutability for safe reuse. Although the execution model differs, the underlying principle of dependency-aware reuse aligns with static scheduling and register allocation in flattened logic graphs. [12]

This study examines early intermediate representations used in compiler construction, emphasizing their role in simplifying optimization and code generation. The authors argue that early normalization of control flow and expressions reduces later-phase complexity. By transforming programs into structured IRs with explicit dependencies, compilers can perform global optimizations more effectively. The paper highlights expression trees, DAGs, and basic block graphs. Although hardware execution is not considered, the work establishes foundational principles for representation-driven optimization. The emphasis on early flattening and dependency exposure strongly supports compiler pipelines that eliminate syntactic constructs early. This paper serves as a conceptual precursor to modern logic-centric compilation approaches, where computation is represented as explicit dependency graphs rather than sequences of statements. [13]

This paper explores instruction scheduling techniques for VLIW and statically scheduled architectures. The authors present algorithms that exploit known latencies and functional unit constraints to generate efficient schedules. By resolving hazards at compile time, the approach eliminates the need for dynamic scheduling hardware. Experimental evaluations demonstrate improved performance and reduced energy consumption. However, the work assumes instruction-based execution models and variable registers. Control flow remains intact, and scheduling is performed after instruction selection. The paper strongly supports static scheduling philosophies. It shows that compile-time resolution of execution order and resource usage is both feasible and efficient when the target architecture is simple. These principles directly inform virtual hardware execution models where scheduling is fully static and deterministic. [14]

This work investigates the use of hardware synthesis techniques in software compilation, particularly the translation of high-level descriptions into register-transfer-level representations. The authors demonstrate that synthesizable models enable aggressive optimization and parallel execution. The paper bridges the gap between software compilation and hardware design, emphasizing dataflow-driven execution. However, the synthesis process assumes specialized tooling and does not generalize to arbitrary software programs. Despite this, the work provides strong motivation for compiler pipelines that treat computation as hardware-like logic rather than sequential instructions. The emphasis on dataflow, registers, and deterministic execution aligns closely with logic-level compilation strategies. The paper highlights the benefits of merging compiler and hardware synthesis concepts to achieve predictable and optimized execution. [15]

This paper focuses on compiler techniques for translating high-level program constructs into low-level representations suitable for hardware acceleration. The authors propose a transformation pipeline that incrementally removes control flow and converts computation into dataflow-oriented structures. Emphasis is placed on preserving semantic correctness while enabling aggressive parallel execution. The paper demonstrates that eliminating conditional branching and mutable state early in the pipeline simplifies downstream optimization. Nevertheless, the methodology strongly supports the concept of flattening programs into explicit dependency graphs. The authors highlight that dataflow representations expose parallelism more naturally than instruction-based models. This insight directly aligns with compiler designs that translate logic into flat boolean or arithmetic circuits. Although register allocation and scheduling are not explored in detail, the paper reinforces the importance of early abstraction removal to enable deterministic execution. [16]

This study examines static scheduling approaches for real-time and safety-critical systems. The authors argue that predictable execution is essential in environments where timing guarantees are required. They propose compile-time scheduling techniques that resolve execution order, resource usage, and memory allocation before deployment. The paper contrasts static scheduling with dynamic approaches, demonstrating that static methods reduce runtime overhead and eliminate unpredictability. While the execution model assumes task-level scheduling rather than instruction-level execution, the underlying philosophy is consistent with deterministic compiler pipelines. The work does not address fine-grained logic execution or register reuse. However, it reinforces the idea that execution decisions should be resolved as early as possible. This principle is fundamental to virtual hardware execution models where logic evaluation order is fixed at compile time. The paper provides strong motivation for static scheduling in constrained execution environments. [17]

This paper explores optimization techniques for dataflow-based execution engines. The authors propose methods for node fusion, dependency pruning, and pipeline balancing to improve throughput. By representing computation as a directed acyclic graph, the execution engine can exploit fine-grained parallelism. The study demonstrates that careful graph optimization significantly reduces execution latency. While the paper focuses on runtime dataflow engines, it highlights the benefits of explicit dependency modeling. The execution remains dynamic, with scheduling decisions made at runtime.

In contrast, static logic compilers aim to resolve these decisions at compile time. Nonetheless, the optimization techniques discussed are directly applicable to static graph scheduling. The work reinforces the value of DAG-based representations for exposing parallelism and enabling aggressive optimization. It provides empirical evidence that graph-centric execution models outperform sequential instruction execution in suitable workloads. [18]

This work investigates compiler-assisted register reuse techniques aimed at reducing storage overhead in data-intensive computations. The authors analyze lifetime overlap and propose strategies for sharing registers among non-overlapping values. By performing global lifetime analysis, the compiler minimizes register count without introducing spills. The study demonstrates that aggressive reuse improves memory efficiency and execution speed. While the approach operates on traditional variable-based IRs, the underlying concept of lifetime-driven reuse maps naturally to logic-level compilation. In flattened logic graphs, values have well-defined lifetimes based on dependency structure. The paper does not address logic synthesis or boolean computation directly. However, it provides important insights into resource minimization strategies. These insights directly inform register allocation phases in virtual hardware pipelines where storage elements are explicitly managed. [19]

This paper presents a modern compiler framework that integrates static analysis, optimization, and code generation into a unified pipeline. The authors emphasize modular design and pass composition to enable extensibility. The framework supports multiple IR levels, allowing gradual lowering from high-level constructs to machine-level code. While flexible, the approach introduces complexity and retains control-flow constructs until late stages. The paper demonstrates that multi-level IR pipelines facilitate experimentation but can obscure global optimization opportunities. In contrast, single-level flattened representations expose all dependencies simultaneously. The work implicitly motivates compiler designs that reduce the number of abstraction layers. Although the execution model remains instruction-based, the framework provides useful insights into organizing large compiler systems. It highlights trade-offs between modularity and global visibility, which are central considerations in logic-centric compiler architectures. [20]

1) Importance of Intermediate Representation Flattening

A central observation across the literature is that the effectiveness of optimization is fundamentally dependent on the structure of the intermediate representation. High-level constructs, control flow, and syntactic abstractions obscure true dependencies, limiting the scope of transformation.

Graph-based representations provide improved visibility; however, they often retain symbolic variables and control structures. In contrast, this work adopts complete flattening of computation into a Boolean dependency graph, eliminating both control flow and mutable state. Computation is reduced to immutable logic expressions, enabling canonical representation and uniform optimization.

The inference is that flattening is not merely an optimization step but a foundational requirement for achieving global correctness, predictability, and aggressive transformation.

2) Static Scheduling for Deterministic Execution

The literature suggests that while dynamic scheduling offers flexibility, it introduces runtime overhead, nondeterminism, and complexity. When dependencies are explicitly known, scheduling decisions can be resolved statically with greater efficiency.

This work adopts a fully static scheduling model, where execution order is determined entirely at compile time using the dependency graph. Unlike traditional compilers that schedule instructions post-generation, scheduling is performed directly on logic operations. This eliminates runtime decision-making and ensures deterministic execution.

The inference is that explicit dependency modeling enables complete elimination of dynamic scheduling without sacrificing performance.

3) Register Allocation through Dependency-Aware Lifetime Analysis

Register pressure is consistently identified as a major bottleneck in both software and hardware execution systems. Effective register reuse requires accurate lifetime analysis and integration with scheduling.

In this work, lifetimes are derived directly from the dependency graph, where each value's existence is determined by its usage within the graph. Unlike variable-based lifetimes dependent on control flow, logic node lifetimes are deterministic and acyclic.

Register allocation is therefore treated as a graph-level optimization problem, enabling efficient reuse and reduced resource consumption. The inference is that integrating allocation with dependency-aware scheduling yields superior results compared to instruction-level techniques.

4) Elimination of Control Flow

A recurring limitation in existing approaches is the presence of control flow, which fragments computation and restricts global optimization. Even advanced techniques rely on inlining or partial simplification while retaining branching constructs. This work extends the inference further by eliminating control flow entirely. Conditional operations are represented as Boolean expressions, allowing uniform treatment of all computation. This aligns with hardware design principles, where execution is governed by data dependencies rather than procedural flow.

The inference is that removing control flow exposes maximum parallelism and simplifies both optimization and scheduling.

5) Transition to Hardware-Inspired Execution

The literature reflects a gradual convergence between compiler design and hardware execution models. Dataflow systems and hardware synthesis approaches demonstrate the advantages of dependency-driven computation.

This work fully adopts this paradigm by introducing a virtual hardware execution model. Computation is expressed as scheduled logic operations over registers, mirroring hardware semantics. Execution becomes evaluation of logic rather than instruction dispatch. The inference is that optimal performance is achieved not by refining software abstractions but by replacing them with hardware-inspired computation models.

B. Identified Research Gaps and Limitations

The reviewed literature provides extensive coverage of compiler intermediate representations, optimization strategies, instruction scheduling, register allocation, and dataflow execution models. However, despite the depth and breadth of these works, a clear research gap emerges when considering end-to-end compilation pipelines that fully eliminate software-centric abstractions and adopt hardware-inspired execution semantics. Most existing papers operate within incremental refinement of traditional compiler architectures. Even when advanced techniques such as SSA, graph-based IRs, or static scheduling are employed, control flow, mutable variables, and procedural abstractions remain embedded within the system. These elements inherently restrict global visibility of computation and complicate lifetime analysis, scheduling, and resource reuse. Several works attempt partial solutions through aggressive inlining, interprocedural optimization, or dataflow execution engines. However, these approaches either retain runtime scheduling decisions or rely on specialized execution environments. The literature does not present a unified pipeline that begins with a custom language designed for bit-level logic expression and ends with a deterministic, statically scheduled virtual hardware execution model. Furthermore, existing research treats scheduling, register allocation, and optimization as loosely coupled or sequential phases, leading to suboptimal global decisions. The proposed project addresses this gap by introducing a fundamentally different compilation philosophy. Instead of lowering high-level constructs gradually, it immediately flattens computation into a logic-level dependency graph. This representation removes control flow entirely, enforces immutability, and exposes complete dependency information. Scheduling and register allocation are performed directly on this graph, ensuring deterministic execution and optimal resource reuse.

In essence, the research gap lies in the absence of a compiler that treats software as hardware from the outset. The proposed project fills this gap by unifying logic representation, static scheduling, register reuse, and execution into a single coherent pipeline, advancing beyond the incremental optimizations explored in existing literature.

C. Derived Objectives for Proposed Work

Existing approaches to bit-level computation lack a balance between clarity, optimization, and execution efficiency. Most implementations rely on general-purpose programming languages, where bit manipulation is performed through low-level operations such as masking, shifting, and indexing. While these techniques are flexible, they obscure the structure of computation, making programs difficult to analyze and optimize. On the other hand, hardware-oriented approaches provide efficiency but lack flexibility and are not easily adaptable for general-purpose software development. This creates a gap between expressiveness and performance in bit-level processing.

To address these limitations, this work proposes a new paradigm where bit-level logic is expressed explicitly and transformed into a dependency-driven computational model. Instead of treating programs as sequences of instructions, the system represents them as structured networks of Boolean operations, enabling global optimization and deterministic execution. The LOCKE framework is designed to shift computation from control-driven execution to logic-driven evaluation, allowing efficient and scalable processing of bit-level operations.

The goals of this research are :

1) To introduce a domain-specific language that enables direct and explicit expression of bit-level operations such as slicing, concatenation, and indexing, improving clarity and correctness.
2) To transform high-level programs into a flattened Boolean dependency graph that captures all computational relationships, eliminating ambiguity and enabling global analysis.
3) To design optimization techniques such as constant folding, redundancy elimination, and structural simplification to reduce computational complexity and improve efficiency.
4) To develop a static scheduling mechanism that organizes operations into deterministic execution stages based on dependency analysis, enabling parallel execution while eliminating runtime overhead.
5) To enable efficient execution through a register-based model and Boolean ALU abstraction, allowing translation into optimized executable code.
6) To support multiple bit-width configurations and vectorized execution strategies, improving performance and scalability across different hardware environments.
7) To exploit both data-level and thread-level parallelism, enabling high throughput and efficient utilization of modern multi-core systems
8) To ensure deterministic and predictable execution by resolving all dependencies and execution order at compile time.

All of these objectives collectively aim to establish a unified framework for bit-level logic processing, overcoming the limitations of traditional programming models while combining the efficiency of hardware-inspired execution with the flexibility of software systems.

III. SYSTEM DESIGN & ARCHITECTURE

A. BITSMITH Frontend Design

The BITSMITH module serves as the frontend of the LOCKE framework and is responsible for processing the input program and constructing its structural representation. This stage focuses on syntactic and structural correctness without introducing any execution semantics, thereby producing a clean and hardware-agnostic model of the program.data units.

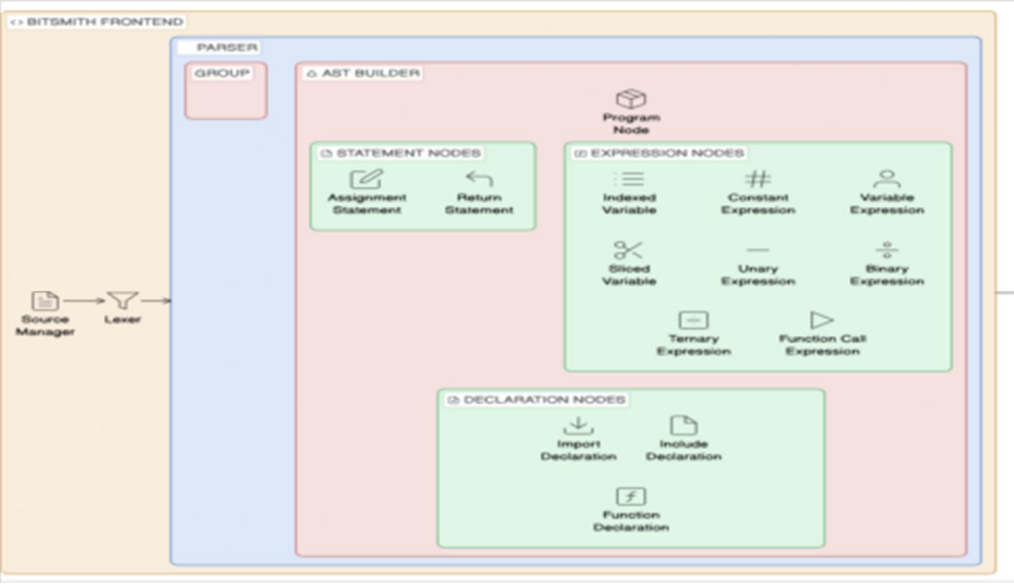


Fig. 1. Architecture of BITSMITH

As shown in Fig 1., BITSMITH operates through a sequence of components, beginning with the source manager, which handles input reading and preprocessing. The lexer then processes the raw input and converts it into a sequence of tokens, identifying elements such as identifiers, operators, and literals. These tokens are passed to the parser, which validates the syntax of the program based on predefined grammatical rules. The parser ensures that the program adheres to the expected structure and constructs a hierarchical representation of the logic.

The AST builder organizes the parsed elements into an Abstract Syntax Tree, where each node represents a specific operation or expression. This tree structure captures the relationships between different components of the program, providing a clear and structured representation of computation.

B. FORGE Logic Construction and Optimization

The FORGE module shown in Fig 2., represents the core transformation stage of the LOCKE framework, where high-level program structure is converted into a logic-centric computational model. This stage is responsible for both constructing the Boolean dependency graph and applying a series of optimizations to improve efficiency.

The transformation begins with bit blasting, a process in which high-level operations are decomposed into fundamental Boolean primitives. For example, arithmetic operations such as addition are expressed as combinations of XOR, AND, and carry propagation logic. Similarly, comparison and logical operations are broken down into basic gate-level representations. This ensures that all computations are represented in a uniform and unambiguous manner. Once the operations are decomposed, the graph constructor builds a directed acyclic graph (DAG) that represents the dependencies between operations. In this graph, each node corresponds to a Boolean operation, and edges represent data dependencies. This structure makes all relationships explicit, allowing the system to analyze and optimize the computation globally.

C. ANVIL Scheduling Design

The process begins by analyzing the dependency structure of the graph. Using topological sorting, the system determines an ordering of operations that satisfies all dependencies. However, instead of producing a purely sequential order, ANVIL groups operations into stages based on their independence.As seen in Fig 3., each stage consists of a set of operations that can be executed simultaneously, as they do not depend on one another. By organizing operations into such stages, the system effectively transforms the graph into a layered execution model.

This scheduling approach has several advantages. First, it minimizes execution depth by reducing the number of sequential steps required to complete the computation. Second, it maximizes parallelism by identifying all operations that can be performed concurrently. Third, it ensures determinism, as the execution order is fully defined at compile time.

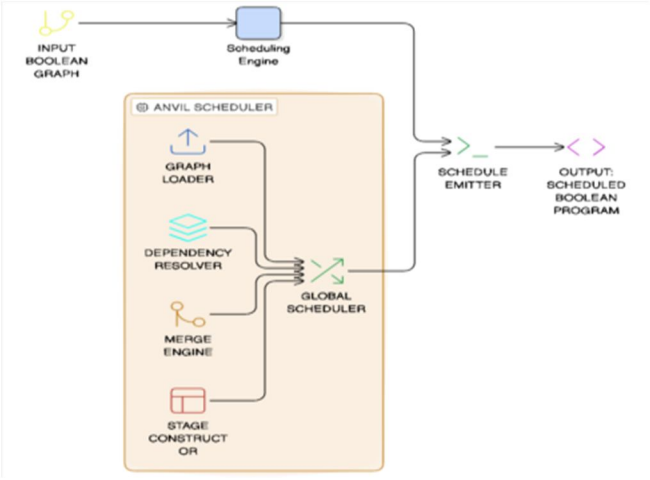


Fig. 2. Architecture of FORGE

After constructing the graph, the logic optimizer applies a series of transformations to reduce complexity and improve efficiency. These include constant folding, where constant expressions are evaluated at compile time; redundancy elimination, which removes duplicate computations; and common subexpression elimination, which identifies and reuses repeated patterns. The result is a compact and optimized representation of the computation that minimizes unnecessary operations.

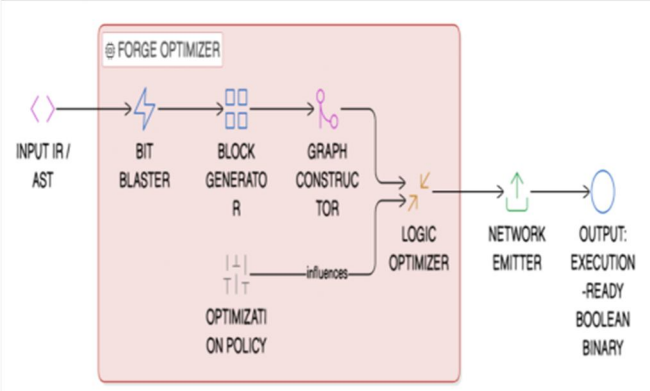


Fig. 3. Architecture of ANVIL

By resolving all scheduling decisions before execution, ANVIL eliminates runtime overhead associated with dependency checking and task management. The result is a predictable and efficient execution plan that is well-suited for both scalar and parallel processing environments.

D. CAST Execution Model

CAST module is the implementation manager allowing the translation of the scheduled computation into executable code without altering structure and dependencies.

```
C sample64.h > ...
#ifndef GEN_64_H
#define GEN_64_H
#include <stdint.h>

static inline void sample_vertical_64(uint64_t* inputs, uint64_t* outputs) {
    uint64_t r[174];
    r[0] = 0;
    r[1] = ~(uint64_t)0;
    r[2] = inputs[0];
    r[3] = inputs[1];
    r[4] = inputs[2];
    r[5] = inputs[3];
    r[6] = inputs[4];
    r[7] = inputs[5];
    r[8] = inputs[6];
    r[9] = inputs[7];
    r[10] = inputs[8];
    r[11] = inputs[9];
    r[12] = inputs[10];
    r[13] = inputs[11];
    r[14] = inputs[12];
    r[15] = inputs[13];
    r[16] = inputs[14];
    r[17] = inputs[15];
```

Fig. 4. Architecture of CAST

In Fig 4., CAST simulates a simplified runtime environment simulating a virtual processor. This environment is made up of a collection of registers to hold intermediate values, a Boolean arithmetic logic unit (ALU) to do operations, and a controller to control the flow of execution. The program that is to be executed is loaded into this environment and the execution occurs stage by stage as per the set schedule.

Every operation of a scheduled graph is translated to an instruction that is executed on registers. Once all the dependencies are already solved, then no runtime decision-making is needed, and the execution has a simple and deterministic pattern. The last phase of the pipeline is to produce executable code according to the computation that is set to take place. The code generated is usually C code and it is represented using a register array to indicate intermediate values. The generated code is a sequence of Boolean operations, one in each line, with the operands being represented at indexed register locations. The easiness of such a representation provides efficient implementation and straightforward mapping to hardware instructions.

There are various execution configurations that are supported by the system. Operation in scalar mode is done using standard 64-bit data types, with compatibility with general-purpose systems. When operating in a vectorized mode, there are multiple bits being processed with SIMD instructions, which greatly enhances throughput. This flexibility in executable strategy results in LOCKE demonstrating high performance on a broad platform of hardware systems.

In Fig 5., header sample64.h Code contains an implementation of the generated Boolean circuit on standard uint64-t registers. The virtual register file is represented by array r [174] and is filled with inputs, constants and all the intermediate logic nodes. Initializing constant 0 and constant 1 (all bits set) is the initial few entries, then the primary inputs are mapped into fixed register slots. The successive lines make a deterministic bit-wise operation (such as XOR) and each assignment relates to a gate in the generated Boolean net. This is a scalar, combinational execution of your compiled circuit on 64 bit words.

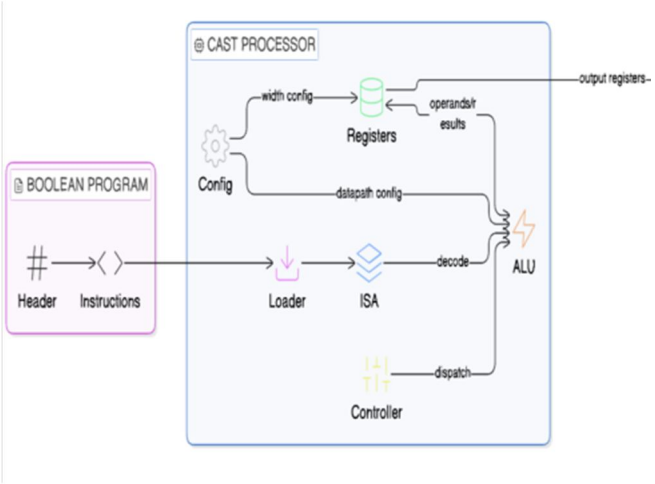


Fig. 5. CAST Output 64-bit (code snippet)

In Fig 6., the sample256.h would be the same logic but with AVX2 SIMD intrinsics (__m256i 256 bits) used to handle 256 bits at once. Once again, the register array was the virtual logic nodes, and loads and XORs are accomplished with the help of _mm256_load_si256 and _mm256_xor_si256. The circuit identifies a number of data lanes in one targeted AVX2, which dramatically enhances throughput without affecting the logical behavior. A vectorized implementation of the same binary action plan.

C sample256.h > ...

```
#ifndef GEN_H
#define GEN_H
#include <immintrin.h>
#include <stdint.h>
#include <string.h>

__attribute__((target("avx2")))
static inline void sample_vertical_256(__m256i** inputs, __m256i** outputs) {
    __m256i r[174];
    r[0] = _mm256_setzero_si256();
    r[1] = _mm256_set1_epi32(-1);

    r[2] = _mm256_load_si256(inputs[0]);
    r[3] = _mm256_load_si256(inputs[1]);
    r[4] = _mm256_load_si256(inputs[2]);
    r[5] = _mm256_load_si256(inputs[3]);
    r[6] = _mm256_load_si256(inputs[4]);
    r[7] = _mm256_load_si256(inputs[5]);
    r[8] = _mm256_load_si256(inputs[6]);
    r[9] = _mm256_load_si256(inputs[7]);
    r[10] = _mm256_load_si256(inputs[8]);
    r[11] = _mm256_load_si256(inputs[9]);
    r[12] = _mm256_load_si256(inputs[10]);
    r[13] = _mm256_load_si256(inputs[11]);
    r[14] = _mm256_load_si256(inputs[12]);
    r[15] = _mm256_load_si256(inputs[13]);
    r[16] = _mm256_load_si256(inputs[14]);
    r[17] = _mm256_load_si256(inputs[15]);
```

Fig. 6. CAST Output 256-bit (code snippet)

IV. SYSTEM IMPLEMENTATION

A. BITSMITH : Parsing and AST Construction

BITSMITH (BITS Manipulation In Translatable High-level Language) module does the conversion of the textual input program to a structured and semantically meaningful representation by an Abstract Syntax Tree (AST). This phase ensures that the input program is syntactically correct as well as structured into a hierarchical form that correctly represents the logical relations of the operations, variables and control structures. It is implemented as a front-end based on a classical compiler design except that, rather than a single generic node type, polymorphic class hierarchy is used to represent the various types of AST nodes.

It starts with the lexical analysis, during which the lexer reads the input stream of characters and translates them into a token stream. These tokens are basic tokens, like identifiers, numeric constants, hexadecimal numbers, operators, delimiters, and keywords such as return, import, and function. The tokens also have positional metadata like the line and column number and this is used at a later stage to report errors during parsing.

To build the AST, the parser, a C++ implementation in parser.cpp, adopts the recursive descent approach to parsing with operator precedence parsing. Contrasting with simpler designs that use only one ASTNode type, this system introduces nodes hierarchy using base classes like Expr, Stmt and Decl, and special derived classes like BinaryExpr, UnaryExpr, ConstExpr, VariableExpr, CallExpr, AssignStmt and ReturnStmt. Each node captures the operation being executed as well as pointers to its child nodes via smart pointers (std::unique_ptr), which guarantee good memory management and ownership semantics.

The processing of expression parsing is a mixture of the functions of parsePrimary and parseExpr. The most simple units of computation (including constants, variables, unary operations and even greater complexity: calls to functions) are recognized by the definePrimary() function. ParseExpr which is used to deal with more complex expressions employs a recursive mechanism, based on precedence, to properly interpret binary and ternary operations. As an example, parsing binary expression will initially form the left hand expression, then recursively build the right hand expression in the way of operator precedence after which a BinaryExpr object will be formed, connected with the left hand and right hand expression objects. This makes sure that a + b * c is properly understood based on the rules of precedence.

It has also elaborate representations of variables, such as indexed access and slicing, via classes such as IndexedVarExpr and SlicedVarExpr. It also deals with the function calls (CallExpr) and conditional expression (TernaryExpr), which enables the AST to model a broad set of program constructs. Languages are broken down into statements with statements being processed individually, such as functions such as a parseAssign and a parseReturn, creating the AssignStmt and ReturnStmt respectively. This is a set of statements in a block of the declarations of functions (FuncDecl), the structure of the whole program in the form of the Program object.

B. FORGE : Boolean Graph Construction and Optimization

FORGE (Forge Optimizes and Reduces Gate-level Expressions) module The heartening execution device of the LOCKE framework, the high-level Abstract Syntax Tree produced by BITSMITH is converted into a low-level Boolean dependency graph. This transformation is an important one in that it converts abstract program logic into a form that can be optimized systematically and implemented. Compared to the AST, the Boolean graph does not maintain a syntactic tree, but rather explicitly describes the data dependencies and bit operations, allowing both fine grain control of computation.

The core of this module is the Value structure that describes every node of the Boolean graph. All operations, including those of the constants, logic gates, arithmetic operations, and function calls, get encoded as Value nodes. Maybe this is a bit imprecise, however, each node contains the type of operation (Op) and the references to its input nodes (op1, op2). This referencing scheme, by index, allows the graph to be small and efficiently traversed into a Directed Acyclic Graph (DAG) with each node only relying on the nodes that have been computed before.

Initial processing of all functions starts at forge main.cpp: here, the functions are handled by the build() pipeline. Money Before it builds the graph, the system uses pruneFunction to eliminate dead code. This step scans the body of the functions in the reverse order and only those variables that are contributing to the final output are identified. This causes needless calculations to be eliminated at an early stage, shrinking the size of the graph.

In processFunction(), the actual graph is built where expressions are resolved to node indices and a vector is created to store this. A mapping (visible_vars) of variables named identically to their renditions in bit-level form is used to track the variables. The emit() function manages node creation, and performs simple deduplication with a cache to prevent re-computing identical operations.

Following the construction, the graph is then to be handed over to the Optimizer where simplification and refinement are carried out severally. The initial degree of optimization uses simple rules of Boolean:

- Constant folding (e.g., AND(1, x) → x)
- Idempotent laws (x AND x → x)
- Complement laws (x AND NOT x → 0)
- Double negation elimination

Having constructed the Boolean graph, it is submitted to the Optimizer, which simplifies and optimizes the graph in a series of runs. This is aimed at minimizing the number of redundant computations and creating a more efficient representation of the logic. It begins with simple Boolean simplifications, including constant folds (e.g. operations with 0 and 1), elimination of redundant operands (x AND x `AND x) or inconsistencies (x AND NOT x x AND) and simplification of two negations.

Transformations with further optimization are induced by the optimizer. Methods such as the absorption law eliminate superfluous expressions (e.g., x OR (x AND y) x), and associative flattening uses more concise forms of operations that get deeply nested forms. Simplification XOR also eliminates redundancy through a cancellation of duplicate operands. The system has common subexpression elimination to prevent recomputation in which the same operation is reused rather than rewritten. This decreases the size of the graph as well as the computation cost. One fairly recent development applied in this case is symbolic Shannon expansion, and such variables are temporarily replaced by 0 and 1 to test their effect on the output. When a variable is not important to the result, it is dropped resulting in additional simplification.

C. ANVIL : Scheduling and Dependency Resolution

The ANVIL (Abstract Networks to Virtual Instructions Layer) module does the conversion of the optimized Boolean graph generated by FORGE into a useful execution schedule. While the FORGE stage focuses on what needs to be computed, ANVIL determines how and in what order these computations should be executed. This phase makes sure that it is correct by honoring dependencies and also provides better performance by efficiently scheduling and reusing registers.

It starts with the combination of the function blocks, which are done in anvil_merge.cpp. As functions in the FORGE stage can invoke other functions, ANVIL initially transforms the multi-function structure into a single flattened computation graph. This is recursively implemented by process_block in which the function calls (EXTERN nodes) are inlined and all the nodes are converted into a single block. In the process, a deduplication mechanism is used to reuse the same operations, instead of copying them, and maintain the optimizations made previously.

After getting a flattened graph the scheduler starts by examining the number of times each node is utilized. This is achieved by a use-count array where each node will count the number of times it is used by other nodes. This knowledge is important in determining which computations are not required and register lifetimes in the future.

Unused nodes which are not included in the final product are known as dead nodes. They are removed in a queue based method, with the zero-usage nodes removed successively, and their dependencies revised. This measure will not only guarantee the schedule of execution is not filled with unnecessary operations. The system builds a dependency graph based on adjacency lists after elimination of dead nodes.

This format enables the scheduler to know which of the nodes can be executed and at what point intermediate output can safely be discarded. The real-life scheduling is achieved by a priority queue in which nodes having no dependencies are chosen to be executed. This makes sure that all the necessary inputs of a node are calculated and then it is executed to guarantee accuracy.

In spite of the fact that a complete height-based priority system is specified, the actual implementation mainly guarantees that dependency constraints are observed, and that the scheduling of nodes is executed as soon as they are ready. When a node is processed, the dependent nodes are updated and the new ones that are ready are added to the queue. Dynamic register allocation is one of the main optimizations in ANVIL. Registers are reused wherever feasible (rather than being used one per calculation).

Intermediate values that have a register allocated to them are freed (when they have no further uses) and may be reused in subsequent computations. This will dramatically cut down on the number of registers needed. Each operation is encoded into a VirtualInstruction, a low-level executable operation, as nodes are scheduled. These instructions include:

- Operation type (AND, OR, XOR, NOT)
- Destination register
- Source registers

The entire instructions are stored in a sequential fashion, constituting the final code of execution. Upon completion of scheduling, an AnvilProgram is generated by the system that includes The list of instructions, Total number of registers used and Mapping of outputs to registers. This is a structured output that makes the subsequent stage (CAST) able to directly produce executable code without additional dependency analysis.

D. CAST : Code Generation and Execution Setup

The CAST (Composable Architecture for Semantic Translation) module is responsible for converting the scheduled program produced by ANVIL into fully executable C code. At this stage, all high-level abstractions , such as expressions, graphs, and dependencies have already been resolved. What remains is a direct translation of instructions into low-level, deterministic execution code.

The process begins by reading the compiled binary representation of the program, which includes metadata such as the number of registers, arguments, outputs, and the list of scheduled instructions. This information is extracted using structured reads (e.g., Header, Inst), ensuring that the execution plan generated earlier is preserved exactly.

A key aspect of this stage is the register-based execution model. An array r[] is created to represent all registers required for computation. Two registers are reserved globally: r[0]→ constant 0 and r[1]→ constant 1

All input arguments are mapped into registers starting from r[2] onward, based on the arg_map. This ensures that only the necessary inputs are loaded, avoiding unnecessary data movement. Execution is then performed by iterating through the instruction list. Each instruction is directly translated into a corresponding C operation.

Since the instructions are already scheduled in dependency order, they can be executed sequentially without any runtime checks or reordering, making execution extremely efficient.

An important extension of this module is its support for multiple execution widths. While smaller configurations (like 8, 16, 32, 64-bit) use standard integer operations, larger configurations such as 256-bit leverage SIMD instructions (AVX2) for parallel execution. Despite this variation, the underlying logic remains identical, demonstrating the flexibility of the code generation pipeline.

V. TESTING & RESULTS

A. Test Program Characteristics

The performance benchmarks are based on DSL programs defined in test.txt, as shown in Fig 7., These programs are compiled through the LOCKE pipeline and are specifically designed to stress different aspects of computation, including arithmetic operations, bitwise logic, conditional execution, and deep dependency chains.

A key aspect of the compiler design is the transformation of conditional constructs into branchless logic using Boolean gating. This eliminates runtime branching and avoids pipeline stalls typically caused by branch misprediction, thereby improving execution efficiency.

test.txt

```
function sample:16 {
    v0 = arg[0:8];
    v1 = arg[8:16];
    v2 = 0x33;
    v3 = 0xCC;

    t0 = (v0 > v1) ? v2 : v3;
    v0 = v0 - t0;
    v1 = v1 + t0;
    d0 = v2 | 0x01;
    v3 = v3 / d0;
    v2 = v2 ^ v3;

    t1 = (v0 < v3) ? v1 : v2;
    v0 = v0 + t1;
    v1 = v1 - 0x11;
    d1 = v0 | 0x01;
    v2 = v2 / d1;
    v3 = v3 + v2;

    t2 = (v1 == v2) ? v3 : v0;
    v1 = v1 - t2;
    v2 = v2 + t2;
    d2 = v3 | 0x03;
    v0 = v0 / d2;
    v3 = v3 ^ v0;
```

Fig. 7. Test Code Snippet

B. Single-Thread Pipeline Performance

The single-threaded execution results, shown in Fig 8., establish the baseline performance of the LOCKE system without any parallelism. The total runtime can be expressed as the sum of three components : Total Runtime = Pack Time + Kernel Time + Unpack Time

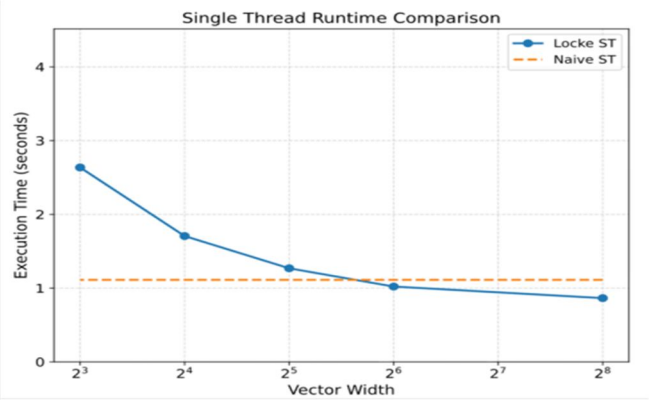


Fig. 8. Single-Thread Pipeline Graph

In contrast, the kernel stage dominates the total execution time. This stage executes all logical and arithmetic operations, including branchless condition handling and register-level instruction chains. At smaller bit widths such as 8 or 16 bits, the system requires a larger number of instructions due to limited parallelism, leading to deeper dependency chains and reduced throughput. As the bit width increases, more data is processed per instruction, reducing instruction count, improving register utilization, and increasing instruction-level parallelism. This results in a significant drop in execution time for larger widths. The unpacking stage mirrors the behavior of packing, as it involves bit extraction and reconstruction of scalar outputs. Similar to packing, it remains memory-bound and contributes a relatively small and stable portion of the total runtime.

C. Multi-Thread Pipeline Performance

The multi-threaded execution results, shown in Fig 9., demonstrate the scalability of the LOCKE pipeline when parallelism is introduced. In this configuration, the input dataset is divided into independent segments, with each thread processing a portion of the data. The workload distribution follows a simple model : Workload per thread = Total samples / Number of threads In practice, the results show that single-thread execution typically ranges between approximately 0.9 to 1.7 seconds depending on configuration, while multi-thread execution reduces runtime to approximately 0.22 to 0.54 seconds in optimized setups. This corresponds to speedups exceeding 5× compared to naive scalar implementations, demonstrating efficient workload distribution and minimal parallel overhead.

D. Percentage Gain Analysis

The percentage gain graph, shown in Fig 10., illustrates the performance improvement achieved by optimized configurations relative to baseline execution.

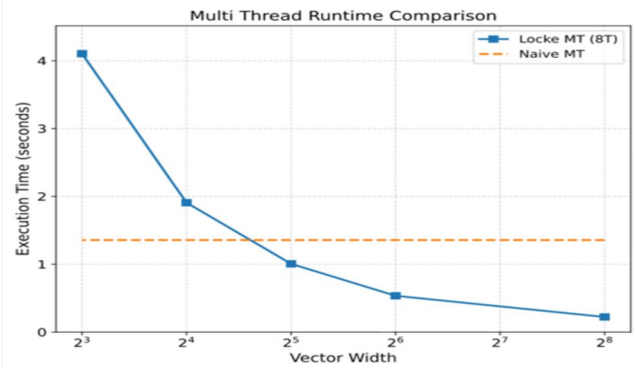


Fig. 9. Multi-Thread Pipeline Graph

The results show that the kernel stage scales very efficiently under multi-threading. This is because the computation is embarrassingly parallel, with no dependencies between threads. Each thread operates on its own data segment, minimizing synchronization overhead and avoiding contention. As a result, the system approaches near-ideal parallel scaling.

The packing and unpacking stages benefit less from multi-threading, primarily due to memory bandwidth limitations. As multiple threads attempt to access memory simultaneously, the memory subsystem becomes a bottleneck. Additionally, minor synchronization overhead further limits scaling in these stages. However, since their contribution to total runtime is relatively small, the overall performance gain remains significant.

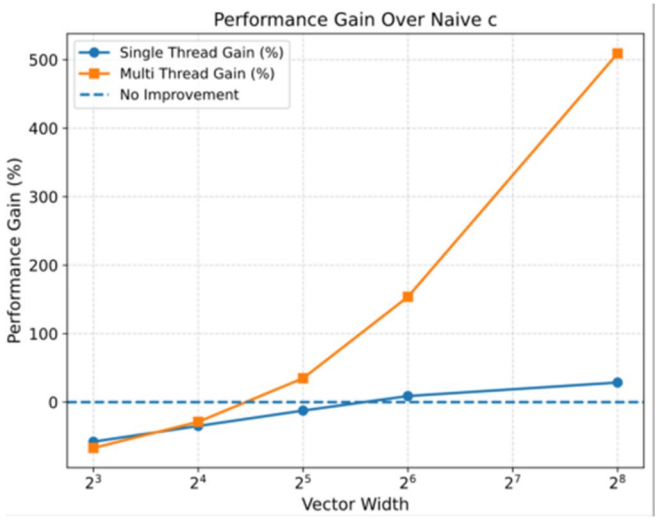


Fig. 10. Time taken to encode the data in Dataset-2

The results indicate that performance gains increase with larger bit widths and are highest when both compiler optimizations and multi-threading are enabled. At smaller widths, the improvement is less pronounced because instruction overhead dominates execution time.

This behavior can be explained by the combined effect of optimization techniques. The compiler reduces redundant instructions, wider data representations increase parallelism, and multi-threading amplifies computational throughput. As workloads become larger and more complex, these optimizations contribute more significantly, leading to higher percentage gains.

E. Throughput Analysis

The throughput graph in Fig 11., measures the amount of data processed per unit time under different configurations.

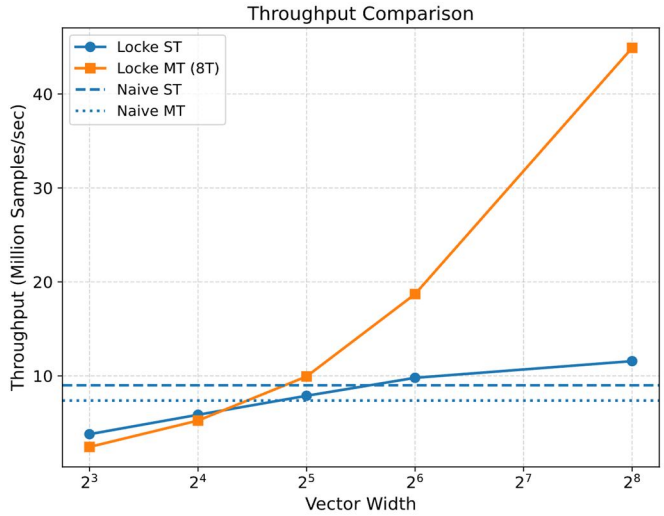


Fig. 11. Throughput Graph

The results show a clear increase in throughput with increasing bit width. Multi-threaded execution further amplifies throughput, while optimized kernels consistently outperform naive implementations.

This improvement is primarily due to the ability to process more bits per operation at higher widths, combined with a reduction in instruction count and efficient distribution of workload across multiple cores. At higher optimization levels, throughput becomes increasingly limited by memory transfer rates rather than computation, indicating that the system is effectively utilizing available compute resources.

F. SpeedUp Analysis

The speedup graph, shown in Fig 12., compares optimized execution against naive baseline implementations.

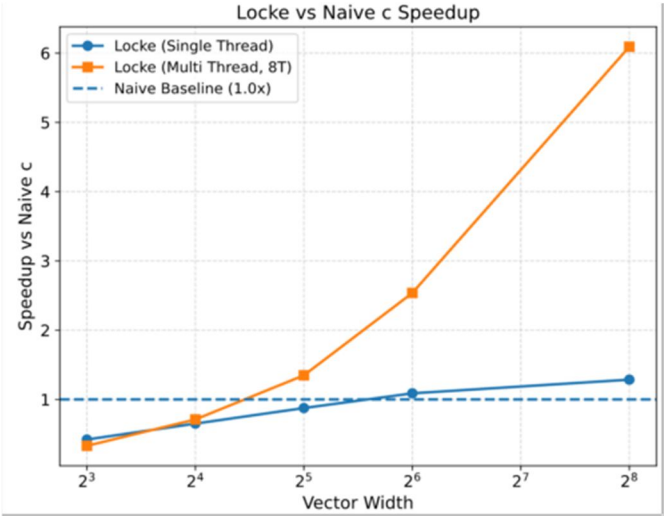


Fig. 12. SpeedUp Graph

The results demonstrate that speedup increases with data width and is highest under multi-threaded configurations. Even in single-thread mode, optimized kernels outperform baseline implementations due to improvements in instruction efficiency and elimination of redundant computation.

The observed speedup is a result of several factors, including Boolean logic simplification, removal of unnecessary operations, elimination of branching, and effective parallel execution. However, speedup tends to plateau at higher configurations due to limitations such as memory bandwidth and thread scheduling overhead. Overall, this analysis highlights how the combination of compiler-level optimizations and hardware-level parallelism contributes to substantial performance improvements in the LOCKE system.

VI. CONCLUSION

As illustrated in its experimental study and system design, LOCKE clearly offers a major improvement over the traditional methods of execution of both Boolean-thorough and data-intensive computations. The framework attains significant performance improvements in execution speed, throughput and hardware usage by converting high-level logic to optimized, branchless, and register-efficient execution sequences. In all configurations, it is evident that the runtime performance is uniformly improved, in particular, when using broader bit-width execution and multi-threading.

One of the strong points of LOCKE is its end-to-end compilation pipeline which removes pools of inefficiency at various steps in a systematic manner. Starting with Boolean graph optimization and common subexpression elimination, dependency-aware scheduling and register reuse, every stage helps to lower the number of instructions and runs faster. The CAST module also provides a guarantee that these optimizations are converted to highly-efficient C code, making the resulting executable with deterministic and predictable implementation without incurring a runtime dependency-resolution overhead or a runtime branching overhead. Performance analysis points out that LOCKE is data-width and parallel executable. Bit-width sizes larger than small should allow even greater data-level parallelism and multi-threading should provide a kind of near-linear scalability through the embarrassingly parallel nature of the load.

All of these enable the system to scale to high throughput and considerable speedups over naive implementations and is suitable in large-scale and compute-intensive applications. LOCKE can also provide new opportunities in compiler-led execution design beyond their performance. The framework provides efficient and predictable execution by moving complexity out of runtime to compile-time. This renders it especially useful in situations where low latency and predictable performance are paramount, e.g. analytics in real time, simulation, pipelines of high performance data processing.

In the future, LOCKE can have a significant potential in becoming a more scalable and versatile system. As additional improvements in the support of its data types, adaptive code generation, advanced optimizations, and integration with new data and AI ecosystems are achieved, it can go past its area of focus and tackle a broader set of computation issues. Even better, parallel and distributed execution can be improved so that the system can be effective in the large-scale setting.

In conclusion, LOCKE is not just an optimization framework, but a rethinking of how computation can be structured and executed efficiently. Through integrating compiler savvy and hardware-sensitive execution policies, it lays a solid groundwork of the next generation high performance computing systems that require performance and scale.

REFERENCES

[1] Z. Li et al., "Unleashing the Power of Compiler Intermediate Representation to Enhance Neural Program Embeddings," IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9793906

[2] M. Belwal and S. T. S. Bindu, "Intermediate representation for heterogeneous multi-core: A survey," International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA), Bengaluru, India, 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7050496

[3] V. P. Bharadwaj and M. Rao, "Compiler optimization for superscalar and pipelined processors," IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), Mangalore, India, 2016. [Online].Available: https://ieeexplore.ieee.org/document/7806224

[4] Y. Huang, Q. Li, and C. J. Xue, "Minimizing Schedule Length via Cooperative Register Allocation and Loop Scheduling for Embedded Systems," IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Changsha, China, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/6120935

[5] M. G. Valluri and R. Govindarajan, "Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors," International Conference on Parallel Architectures and Compilation Techniques (PACT), Newport Beach, CA, USA, 1999. [Online]. Available: https://ieeexplore.ieee.org/document/807420

[6] D. Barhate and A. K. Sarje, "An approach for pointer optimization using SSA based intermediate representation," International Conference on Recent Trends in Information Technology (ICRTIT), Chennai, India, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/5972428

[7] D. Binkley and M. Harman, "Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms," IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Amsterdam, Netherlands, 2003. [Online]. Available: https://ieeexplore.ieee.org/document/1238046

[8] J. Guo, M. Stiles, Q. Yi, and K. Psarris, "Enhancing the Role of Inlining in Effective Interprocedural Parallelization," International Conference on Parallel Processing (ICPP), Taipei, Taiwan, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/6047195

[9] S. Subha, "A Modified Linear Scan Register Allocation Algorithm," International Conference on Information Technology: New Generations (ITNG), Las Vegas, NV, USA,2009.[Online].Available: https://ieeexplore.ieee.org/document/5070724

[10] V. K. S. Nerella, S. K. Madria, and T. Weigert,"Optimization of Object Queries on Collections Using Annotations for the String Valued Attributes," IEEE Annual Computer Software and Applications Conference (COMPSAC), Kyoto, Japan, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6649843

[11] H. Riener, A. Mishchenko, and M. Soeken, "Exact DAG-Aware Rewriting," Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France,2020.[Online].Available: https://ieeexplore.ieee.org/document/9116379

[12] G. M. Slota and K. Madduri, "Fast Approximate Subgraph Counting and Enumeration," International Conference on Parallel Processing (ICPP), Lyon, France,2013.[Online].Available: https://ieeexplore.ieee.org/document/6687354

[13] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors,"Proceedings of the IEEE, 2001. [Online]. Available: https://ieeexplore.ieee.org/document/964443

[14] C. Yu, M. Ciesielski, M. Choudhury, and A. Sullivan,"DAG-aware logic synthesis of datapaths," ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7544377

[15] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis," Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8715185

[16] P. Z. da Silva, H. C. de M. Senefonte, and W. Attrot,"Interference Graph Dataset for Machine Learning-Based Register Allocation," IEEE Access, 2024.[Online].Available: https://ieeexplore.ieee.org/document/10720063

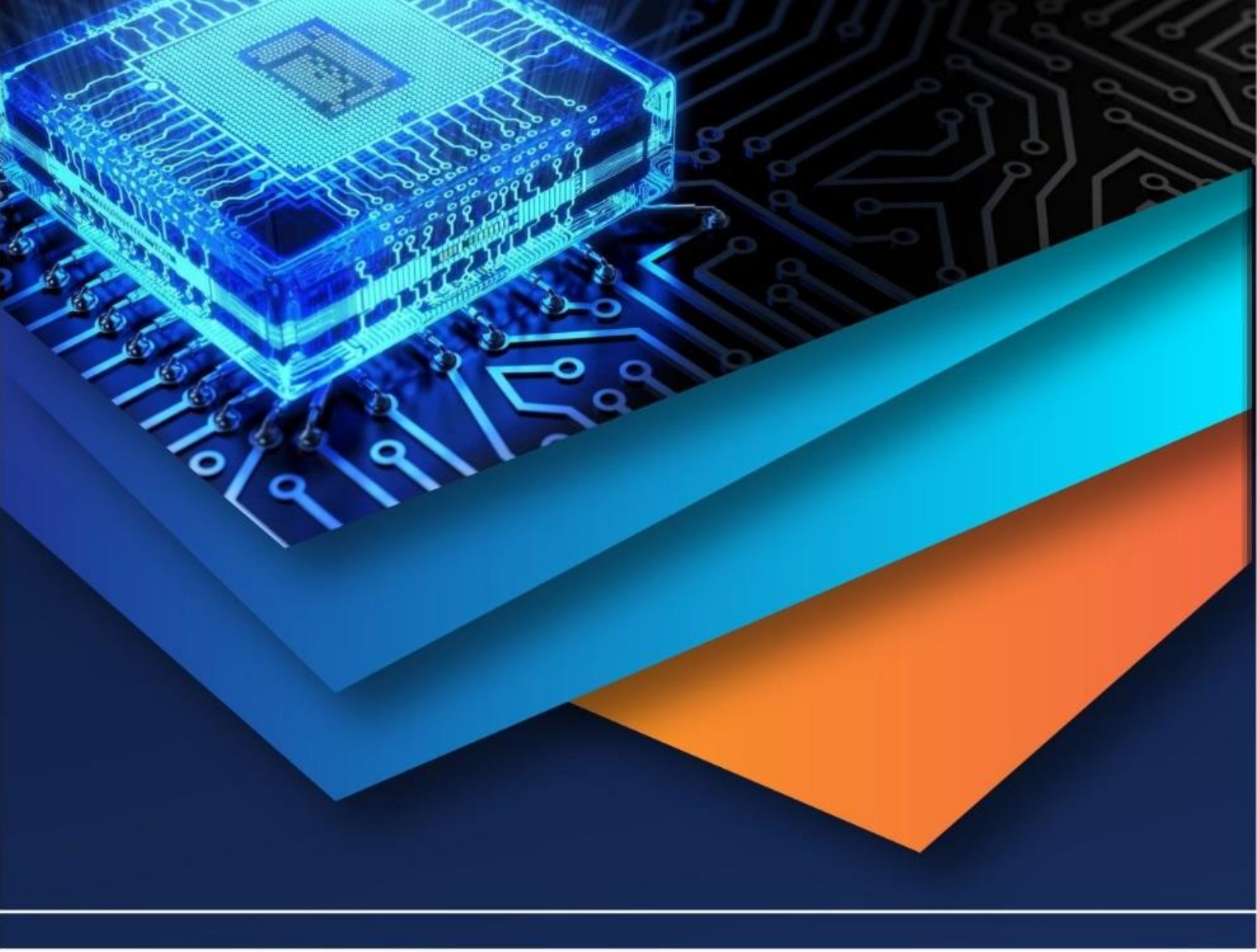
[17] Z. Chu, W. Haaswijk, M. Soeken, Y. Xia, L. Wang, and G. De Micheli, "Exact Synthesis of Boolean Functions in Majority-of-Five Forms," IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 2019. [Online]. Available:

https://ieeexplore.ieee.org/document/8702141

[18] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9378784

[19] T. Nguyen and A. McCaskey, "Retargetable Optimizing Compilers for Quantum Accelerators via a Multilevel Intermediate Representation," IEEE Micro, 2022. [Online].Available: https://ieeexplore.ieee.org/document/10361005

[20] J. Li, Z. Zhang, X. Zhou, and L. Wang, "An MLIR-Based Compiler for Hardware Acceleration with Recursion Support," International Conference on Field Programmable Technology (ICFPT), Sydney, Australia, 2024.[Online].Available: https://ieeexplore.ieee.org/document/11113451



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY