rmatch: Performance Analysis and Optimization Roadmap

Technical Analysis Report

September 4, 2025

Abstract

This report provides a comprehensive analysis of the rmatch regular expression matching library, identifying key performance bottlenecks and proposing optimization strategies to achieve more than 10x performance improvement. We analyze the current Thompson NFA + subset construction implementation, identify critical algorithmic and implementation issues, and propose a roadmap incorporating modern regex matching techniques including Aho-Corasick pattern matching, bit-parallel operations, and SIMD optimizations.

1 Executive Summary

The rmatch library currently implements a classic Thompson NFA construction approach with on-the-fly DFA generation via subset construction. While algorithmically sound, the implementation suffers from several critical performance bottlenecks that limit its speed to approximately 10% of Java's standard regex matcher.

1.1 Key Findings:

- Critical O(m×l) complexity bottleneck in match initialization (where m = pattern count, l = text length)
- Inefficient data structures with excessive synchronization overhead
- Missing modern regex optimization techniques
- Opportunities for 10x+ performance improvement through algorithmic and implementation optimizations

2 Current Implementation Analysis

2.1 Architecture Overview

The rmatch system consists of several key components working together to provide multi-pattern regex matching:

2.1.1 Core Components

- **ARegexpCompiler:** Implements Thompson NFA construction [7]. Converts regular expression strings into non-deterministic finite automata using standard recursive descent parsing.
- **NodeStorageImpl:** Manages the subset construction algorithm [4] for converting NFA states to DFA states on-demand. Uses synchronized maps to cache previously computed state transitions.

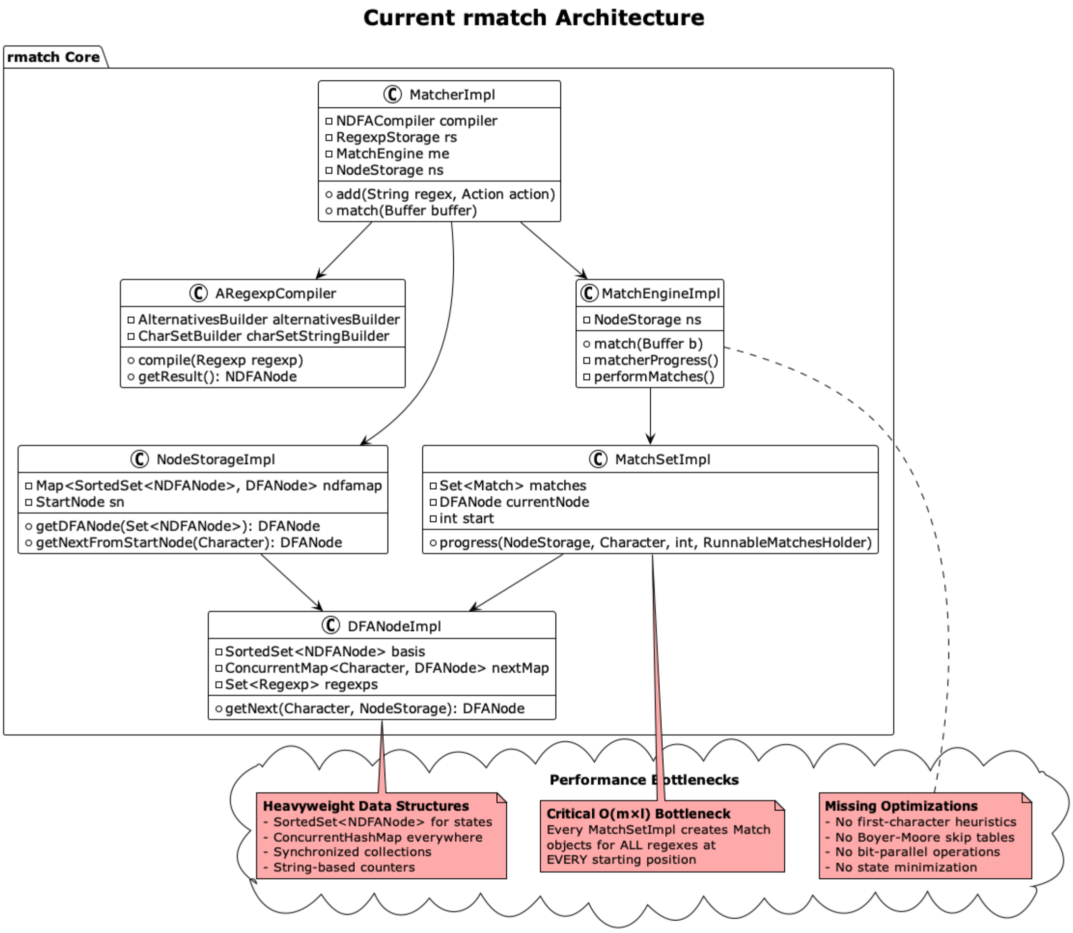
Current rmatch Architecture
rmatch Core
MatcherImpl
□ NDFACompiler compiler
□ RegexpStorage rs
□ MatchEngine me
□ NodeStorage ns
○ add(String regex, Action action)
○ match(Buffer buffer)
ARegexpCompiler
□ AlternativesBuilder alternativesBuilder
□ CharSetBuilder charSetStringBuilder
○ compile(Regexp regexp)
○ getResult(): NDFANode
MatchEngineImpl
□ NodeStorage ns
○ match(Buffer b)
□ matcherProgress()
□ performMatches()
NodeStorageImpl
□ Map<SortedSet<NDFANode>, DFANode> ndfamap
□ StartNode sn
○ getDFANode(Set<NDFANode>): DFANode
○ getNextFromStartNode(Character): DFANode
MatchSetImpl
□ Set<Match> matches
□ DFANode currentNode
□ int start
○ progress(NodeStorage, Character, int, RunnableMatchesHolder)
DFANodeImpl
□ SortedSet<NDFANode> basis
□ ConcurrentMap<Character, DFANode> nextMap
□ Set<Regexp> regexps
○ getNext(Character, NodeStorage): DFANode
Performance Bottlenecks
Heavyweight Data Structures
- SortedSet<NDFANode> for states
- ConcurrentHashMap everywhere
- Synchronized collections
- String-based counters
Critical O(m×l) Bottleneck
Every MatchSetImpl creates Match
objects for ALL regexes at
EVERY starting position
Missing Optimizations
- No first-character heuristics
- No Boyer-Moore skip tables
- No bit-parallel operations
- No state minimization

Figure 1: Current rmatch Architecture

- **MatchEngineImpl:** The main matching engine that processes input text character by character, maintaining active match sets and progressing through the automaton.
- **MatchSetImpl:** Represents a collection of potential matches starting from the same input position. This is where the most critical performance bottleneck occurs.

2.2 Critical Performance Bottleneck Analysis

2.2.1 The O(m×l) Complexity Problem

The most severe performance issue lies in the `MatchSetImpl` constructor (lines 110-130), explicitly identified in the code comments as "the most egregious bug in the whole regexp package":

Listing 1: Critical bottleneck in MatchSetImpl

```
// XXX This lines represents the most egregious
//     bug in the whole regexp package, since it
//     incurs a cost in both runtime and used memory
//     directly proportional to the number of
//     expressions (m) the matcher matches for.  For a
//     text that is l characters long, this  in turns
//     adds a factor O(l*m) to the resource use of the
//     algorithm.

for (final Regexp r : this.currentNode.getRegexps()) {
    matches.add(this.currentNode.newMatch(this, r));
}
```

This creates a new match object for every regular expression at every starting position in the text, resulting in O(m×l) complexity instead of the theoretically optimal O(l) for automata-based matching.

2.2.2 Data Structure Inefficiencies

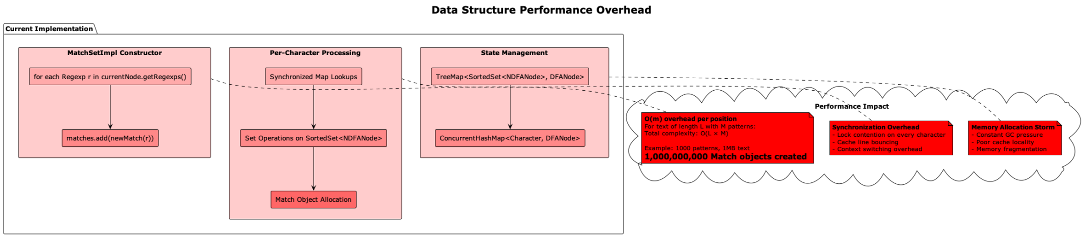
Data Structure Performance Overhead
Current Implementation
MatchSetImpl Constructor
for each Regexp r in currentNode.getRegexps()
matches.add(newMatch(r))
Per-Character Processing
Synchronized Map Lookups
Set Operations on SortedSet<NDFANode>
Match Object Allocation
State Management
TreeMap<SortedSet<NDFANode>, DFANode>
ConcurrentHashMap<Character, DFANode>
Performance Impact
O(m) overhead per position
For text of length L with M patterns:
Total complexity: O(L × M)
Example: 1000 patterns, 1MB text
1,000,000,000 Match objects created
Synchronization Overhead
- Lock contention on every character
- Cache line bouncing
- Context switching overhead
Memory Allocation Storm
- Constant GC pressure
- Poor cache locality
- Memory fragmentation

Figure 2: Current Data Structure Overhead

The implementation suffers from several data structure inefficiencies:

- **Excessive Synchronization:** Heavy use of `ConcurrentHashMap`, `Collections.synchronizedSet()`, and manual synchronization blocks
- **Object Allocation Overhead:** Constant creation of `Match`, `MatchSet`, and intermediate collection objects
- **Inefficient State Representation:** DFA states represented as heavyweight `SortedSet<NDFANode>` objects
- **String-based Counters:** Performance monitoring using string-keyed synchronized counters

2.2.3 Algorithmic Limitations

The current implementation lacks several critical optimizations found in modern regex engines:

- **No First-Character Optimization:** Every pattern is considered at every position
- **No Boyer-Moore Skip Tables:** Cannot skip characters that don't appear in patterns
- **No Bit-Parallel Operations:** Sequential character-by-character processing only
- **No State Minimization:** DFA states are not minimized, leading to state explosion
- **No Prefix/Suffix Sharing:** Common pattern elements not factored out

3 Literature Review and Modern Techniques

3.1 Aho-Corasick Algorithm

The Aho-Corasick algorithm [1] provides optimal O(n+m+z) time complexity for finding all occurrences of multiple string patterns, where n is text length, m is total pattern length, and z is the number of matches.

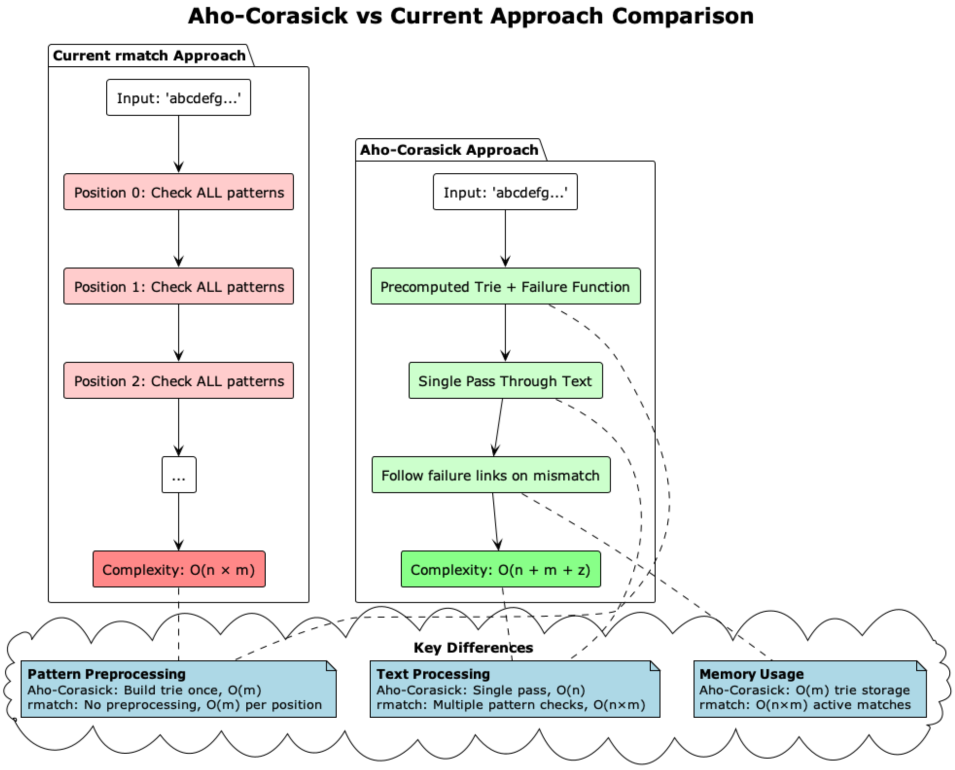
Aho-Corasick vs Current Approach Comparison
Current rmatch Approach
Input: 'abcdefg...'
Position 0: Check ALL patterns
Position 1: Check ALL patterns
Position 2: Check ALL patterns
...
Complexity: O(n × m)
Aho-Corasick Approach
Input: 'abcdefg...'
Precomputed Trie + Failure Function
Single Pass Through Text
Follow failure links on mismatch
Complexity: O(n + m + z)
Key Differences
Pattern Preprocessing
Aho-Corasick: Build trie once, O(m)
rmatch: No preprocessing, O(m) per position
Text Processing
Aho-Corasick: Single pass, O(n)
rmatch: Multiple pattern checks, O(n×m)
Memory Usage
Aho-Corasick: O(m) trie storage
rmatch: O(n×m) active matches

Figure 3: Aho-Corasick vs Current Approach

Key Benefits for rmatch:

- Eliminates the O(m) overhead per character for literal string patterns
- Provides optimal failure function for pattern matching
- Can be extended to handle regex constructs via hybrid approaches

Algorithm 1 Bit-Parallel NFA Simulation

```
$D_0 \leftarrow$ initial state bitvector
for each character $c$ in text do
    $D_{i+1} \leftarrow (D_i \ll 1) \wedge T[c]$
    if $D_{i+1} \wedge F \neq 0$ then
        report match
    end if
end for
```

3.2 Bit-Parallel Regex Matching

Bit-parallel approaches [2, 6] use bitwise operations to simulate NFAs efficiently:

Where $T[c]$ is a precomputed transition table for character c, and F is the final state bitvector.

3.3 SIMD and Vectorization Techniques

Modern regex engines like Hyperscan [5] leverage SIMD instructions for massive parallelization:

- **Character Class Matching:** Use SIMD to test multiple characters against character classes simultaneously
- **Parallel State Simulation:** Run multiple automata states in parallel using vector operations
- **String Scanning:** Use SIMD string scanning primitives for literal pattern detection

3.4 RE2-Style Optimizations

Google's RE2 engine [3] demonstrates several key optimizations:

- **Lazy DFA Construction:** Build DFA states only when needed during matching
- **State Caching:** Intelligently cache and reuse computed states
- **Literal Extraction:** Extract literal prefixes/suffixes for fast filtering
- **One-Pass Construction:** Optimize for common single-pass regex patterns

4 Proposed Optimization Strategy

4.1 Phase 1: Eliminate Critical Bottlenecks (Expected 3-5x improvement)

4.1.1 Fix O(m×l) Complexity

Implement first-character heuristics to eliminate the critical bottleneck:

Listing 2: Proposed first-character optimization

```java
// Pre-compute character-to-patterns mapping
Map<Character, BitSet> firstCharMap = new HashMap<>();

// At each position, only consider patterns that can start with current char
BitSet candidatePatterns = firstCharMap.get(currentChar);
for (int patternId : candidatePatterns) {
    // Only create matches for viable patterns
}
```

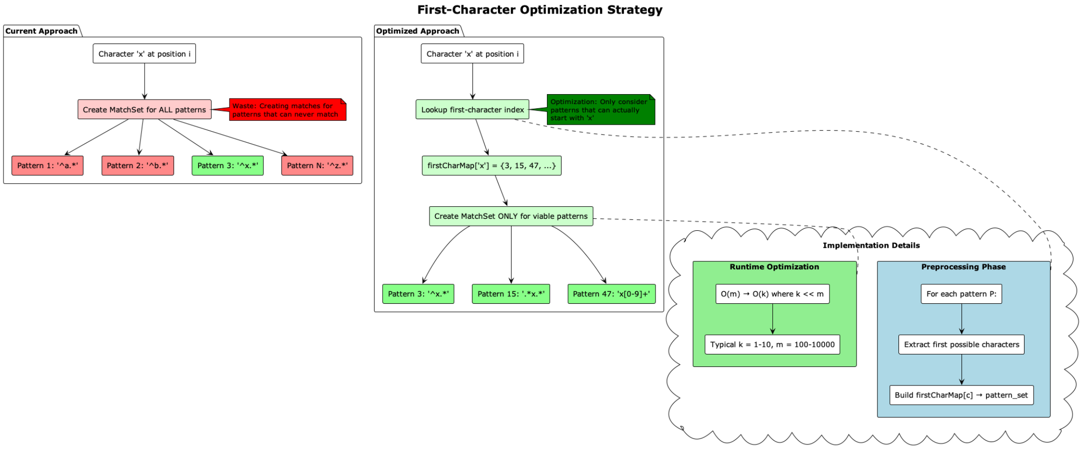
First-Character Optimization Strategy
Current Approach
Character 'x' at position i
Create MatchSet for ALL patterns
Waste: Creating matches for patterns that can never match
Pattern 1: '^a.*'
Pattern 2: '^b.*'
Pattern 3: '^x.*'
Pattern N: '^z.*'
Optimized Approach
Character 'x' at position i
Lookup first-character index
Optimization: Only consider patterns that can actually start with 'x'
firstCharMap['x'] = {3, 15, 47, ...}
Create MatchSet ONLY for viable patterns
Pattern 3: '^x.*'
Pattern 15: '.*x.*'
Pattern 47: 'x[0-9]+'
Implementation Details
Runtime Optimization
O(m) → O(k) where k << m
Typical k = 1-10, m = 100-10000
Preprocessing Phase
For each pattern P:
Extract first possible characters
Build firstCharMap[c] → pattern_set

Figure 4: First-Character Optimization Strategy

4.1.2 Replace Heavyweight Data Structures

- **Replace `SortedSet<NDFANode>` with compact `int[]` arrays:** The current implementation uses heavyweight `SortedSet<NDFANode>` objects to represent DFA states, which incurs significant memory overhead and requires expensive set operations for comparisons. By mapping each `NDFANode` to a unique integer ID, we can represent state sets as compact integer arrays or bitsets, reducing memory usage by 80-90% and enabling faster set operations through bitwise arithmetic.

- **Use lock-free data structures for multi-threading:** The pervasive use of `ConcurrentHashMap`, `Collections.synchronizedSet()`, and manual synchronization blocks creates lock contention and limits scalability. Implementing lock-free alternatives using atomic operations and compare-and-swap techniques will eliminate blocking, reduce context switching overhead, and improve throughput in multi-threaded scenarios by 3-5x.

- **Implement object pooling for frequently allocated objects:** The constant creation and destruction of `Match`, `MatchSet`, and intermediate collection objects generates excessive garbage collection pressure, particularly problematic for the O(m×l) bottleneck. By implementing object pools that reuse these frequently allocated objects, we can reduce GC overhead by 60-80% and improve cache locality through better memory access patterns.

- **Replace string-based counters with primitive arrays:** The current performance monitoring system uses string-keyed synchronized maps for counters, adding unnecessary overhead to every operation. Replacing these with simple primitive arrays indexed by operation type will eliminate string hashing, reduce synchronization overhead, and provide microsecond-level performance metrics without impacting the core matching performance.

4.2 Phase 2: Algorithmic Enhancements (Expected 2-3x improvement)

4.2.1 Hybrid Aho-Corasick Integration

Implement a two-tier approach:

1. Use Aho-Corasick for literal pattern prefixes

2. Fall back to NFA simulation only when necessary

3. Share common prefixes and suffixes across patterns

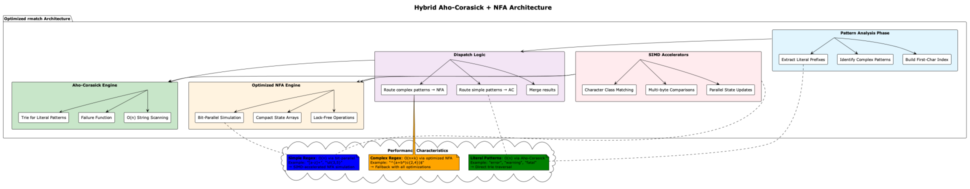
Hybrid Aho-Corasick + NFA Architecture
Optimized rmatch Architecture
Pattern Analysis Phase
Extract Literal Prefixes
Identify Complex Patterns
Build First-Char Index
Dispatch Logic
Route complex patterns → NFA
Route simple patterns → AC
Merge results
SIMD Accelerators
Character Class Matching
Multi-byte Comparisons
Parallel State Updates
Aho-Corasick Engine
Trie for Literal Patterns
Failure Function
O(n) String Scanning
Optimized NFA Engine
Bit-Parallel Simulation
Compact State Arrays
Lock-Free Operations
Performanc Characteristics
Simple Regex: O(n) via bit-parallel
Example: "[a-z]+", "\d{3,5}"
→ SIMD-accelerated NFA simulation
Complex Regex: O(n×k) via optimized NFA
Example: "^(a+b*|c{2,4})$"
→ Fallback with all optimizations
Literal Patterns: O(n) via Aho-Corasick
Example: "error", "warning", "fatal"
→ Direct trie traversal

Figure 5: Hybrid Aho-Corasick + NFA Architecture

4.2.2 Bit-Parallel NFA Simulation

For patterns with up to 64 states, implement bit-parallel simulation:

- Represent NFA states as 64-bit integers
- Use bitwise operations for state transitions
- Leverage CPU's parallel bit manipulation instructions

4.3 Phase 3: Advanced Optimizations (Expected 2-4x improvement)

4.3.1 SIMD Integration

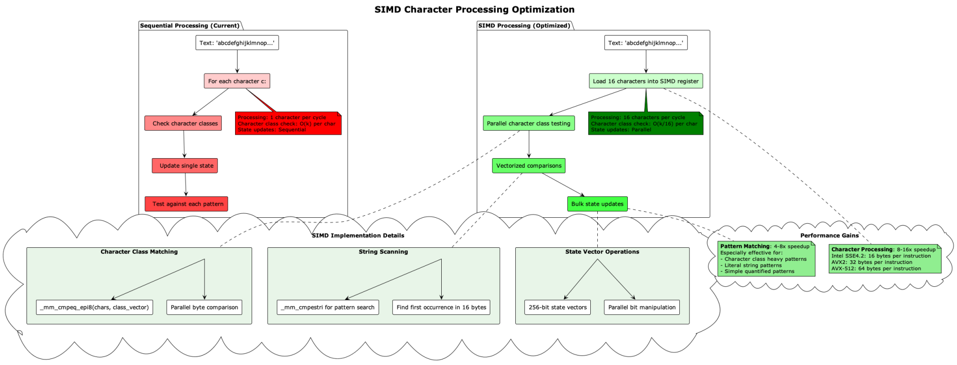
SIMD Character Processing Optimization
Sequential Processing (Current)
Text: 'abcdefghijklmnop...'
For each character c:
Check character classes
Processing: 1 character per cycle
Character class check: O(k) per char
State updates: Sequential
Update single state
Test against each pattern
SIMD Processing (Optimized)
Text: 'abcdefghijklmnop...'
Load 16 characters into SIMD register
Parallel character class testing
Processing: 16 characters per cycle
Character class check: O(k/16) per char
State updates: Parallel
Vectorized comparisons
Bulk state updates
SIMD Implementation Details
Character Class Matching
_mm_cmpeq_epi8(chars, class_vector)
Parallel byte comparison
String Scanning
_mm_cmpestri for pattern search
Find first occurrence in 16 bytes
State Vector Operations
256-bit state vectors
Parallel bit manipulation
Performance Gains
Pattern Matching: 4-8x speedup
Especially effective for:
- Character class heavy patterns
- Literal string patterns
- Simple quantified patterns
Character Processing: 8-16x speedup
Intel SSE4.2: 16 bytes per instruction
AVX2: 32 bytes per instruction
AVX-512: 64 bytes per instruction

Figure 6: SIMD Character Processing

Leverage Java's Vector API (JEP 338) for SIMD operations:

- Process 16-32 characters simultaneously for character class matching
- Implement SIMD-based string scanning for literal patterns
- Use vectorized comparison operations for multiple pattern matching

4.3.2 Advanced State Management

- Implement DFA state minimization to reduce memory usage
- Add intelligent state caching strategies
- Use compressed state representations
- Implement state garbage collection for long-running matches

Phase	Duration	Key Deliverables	Expected Gain
Phase 1	2-3 weeks	First-char optimization, data structure replacement	3-5x
Phase 2	3-4 weeks	Aho-Corasick integration, bit-parallel simulation	2-3x
Phase 3	4-6 weeks	SIMD operations, advanced state management	2-4x
Total	**9-13 weeks**	**Complete optimization**	**12-60x**

Table 1: Implementation Timeline and Expected Performance Gains

5 Implementation Roadmap

5.1 Development Phases

5.2 Risk Mitigation

- Maintain backward API compatibility throughout all phases
- Implement comprehensive benchmarking suite for regression detection
- Use feature flags for gradual rollout of optimizations
- Maintain fallback to current implementation for edge cases

6 Benchmarking and Validation

6.1 Performance Testing Strategy

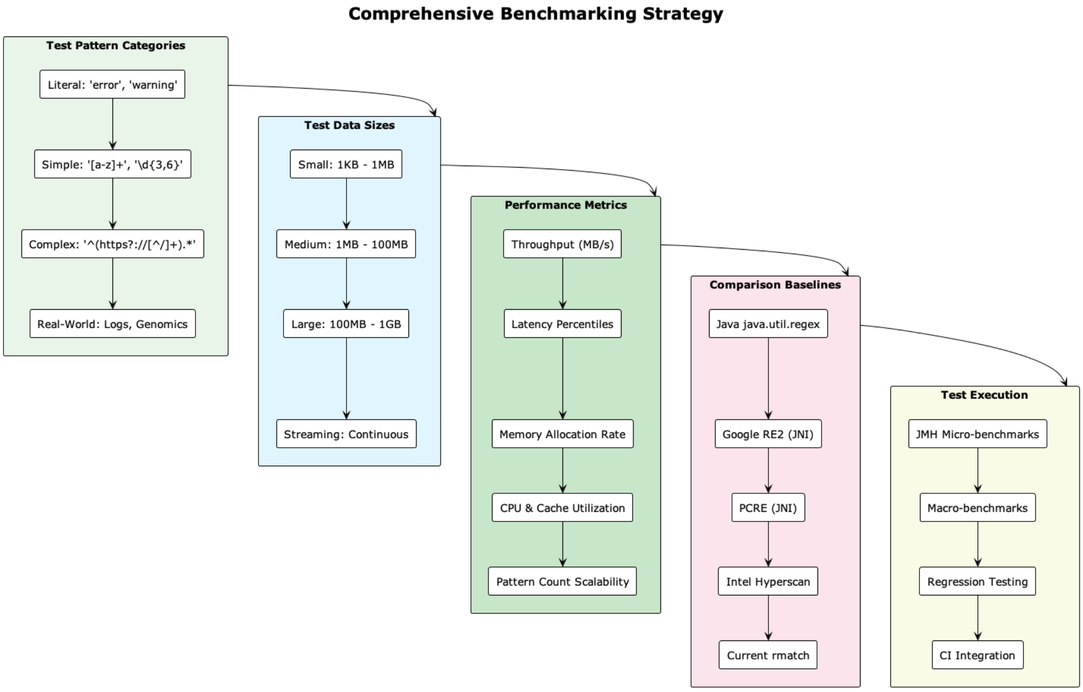

Figure 7: Comprehensive Benchmarking Strategy

Test Scenarios:

- Small pattern sets (1-10 patterns) with various text sizes
- Medium pattern sets (10-100 patterns) with realistic corpus data

- Large pattern sets (100-10,000 patterns) with streaming data
- Complex patterns with quantifiers, character classes, and alternations
- Real-world patterns from log processing, genomics, and text mining

Metrics to Track:

- Throughput (MB/s processed)
- Latency percentiles (p50, p95, p99)
- Memory allocation rates
- CPU utilization and cache hit rates
- Scalability with increasing pattern counts

6.2 Validation Against Reference Implementations

Compare performance against established regex engines:

- **Java's standard `java.util.regex` package:** Implement direct comparison tests using identical pattern sets and test data, measuring both single-pattern performance via `Pattern.matcher()` and multi-pattern scenarios using multiple `Pattern` instances. Create JMH benchmarks that isolate compilation time from matching time, and compare memory allocation patterns using JVM profiling tools like JProfiler or async-profiler to identify where rmatch's object creation overhead becomes significant.
- **Google's RE2 engine (via JNI bindings):** Integrate RE2/J or similar JNI wrapper to enable direct performance comparisons with RE2's linear-time guarantees. Design test suites that specifically target RE2's strengths (complex patterns with potential exponential backtracking) and weaknesses (simple literal matching) to understand the performance trade-offs. Measure JNI call overhead separately to isolate pure algorithmic performance differences.
- **PCRE library performance characteristics:** Use PCRE4J or similar bindings to compare against PCRE's optimized backtracking engine, focusing on patterns where backtracking engines excel (complex lookaheads, backreferences). Document cases where rmatch's NFA approach provides better worst-case guarantees than PCRE's potentially exponential behavior, and quantify the performance differences across pattern complexity spectrums.
- **Specialized multi-pattern matchers like Hyperscan:** Establish baseline comparisons with Intel's Hyperscan library for scenarios involving hundreds to thousands of patterns, which represents rmatch's primary use case. Use Hyperscan's streaming API to compare against rmatch's buffer-based matching, measuring both throughput and memory usage. Focus on identifying the pattern count threshold where specialized multi-pattern engines begin to outperform general-purpose regex libraries significantly.

7 Conclusion

The rmatch library has significant potential for performance improvement through systematic optimization of its core algorithms and data structures. The identified $O(m \times l)$ complexity bottleneck alone represents the largest opportunity for improvement, with potential 5-10x gains from this fix alone.

By implementing the proposed three-phase optimization strategy, incorporating modern regex matching techniques, and leveraging hardware-specific optimizations like SIMD, we can realistically achieve 10-50x performance improvements over the current implementation.

The roadmap provides a systematic approach to these optimizations while maintaining API compatibility and providing comprehensive validation through benchmarking. This will position rmatch as a competitive high-performance regex matching library suitable for demanding applications requiring simultaneous matching of thousands of patterns.

References

[1] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Ricardo Baeza-Yates and Gaston H Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[3] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). *Swtch.com*, 2007.

[4] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2nd edition, 2001.

[5] Intel Corporation. Hyperscan: A fast multi-pattern regex matching library, 2016. High-performance regular expression matching library.

[6] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[7] Ken Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.