

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 $O(m \times l)$ 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.

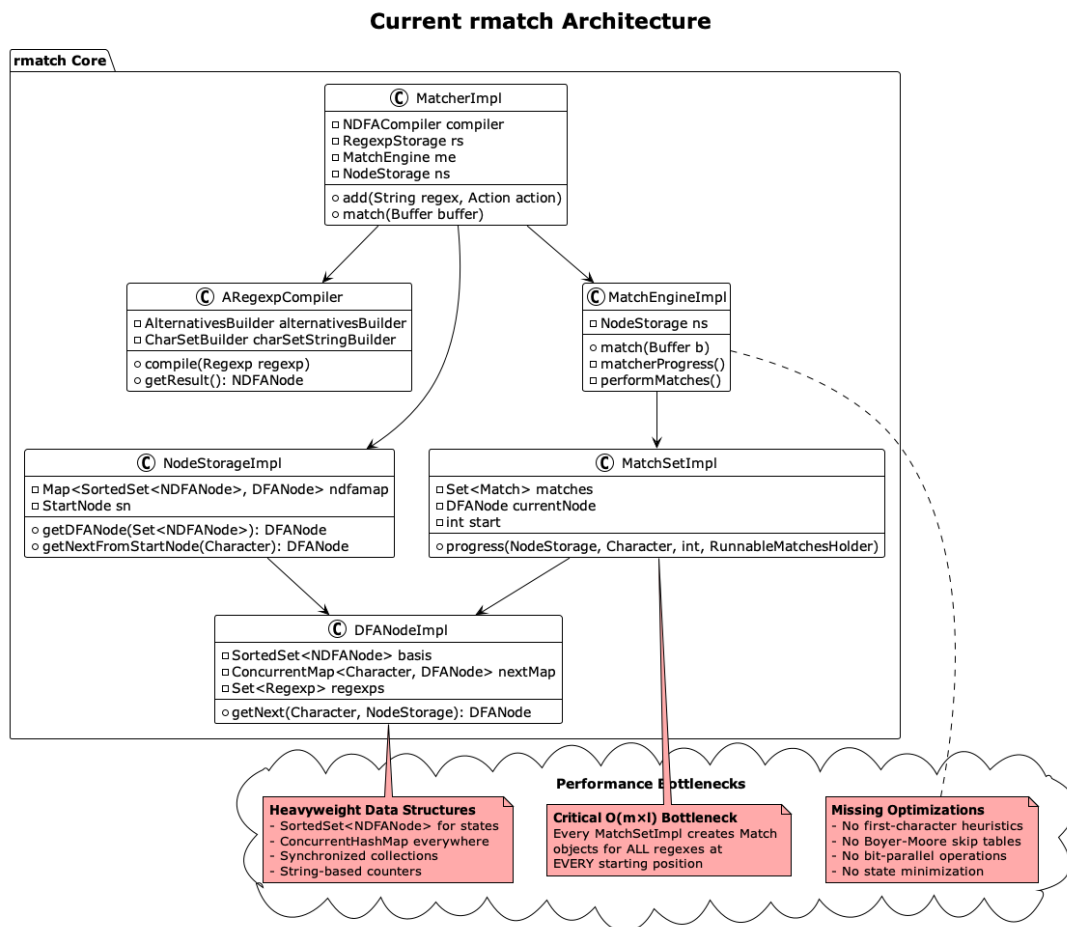


Figure 1: Current rmatch Architecture

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.

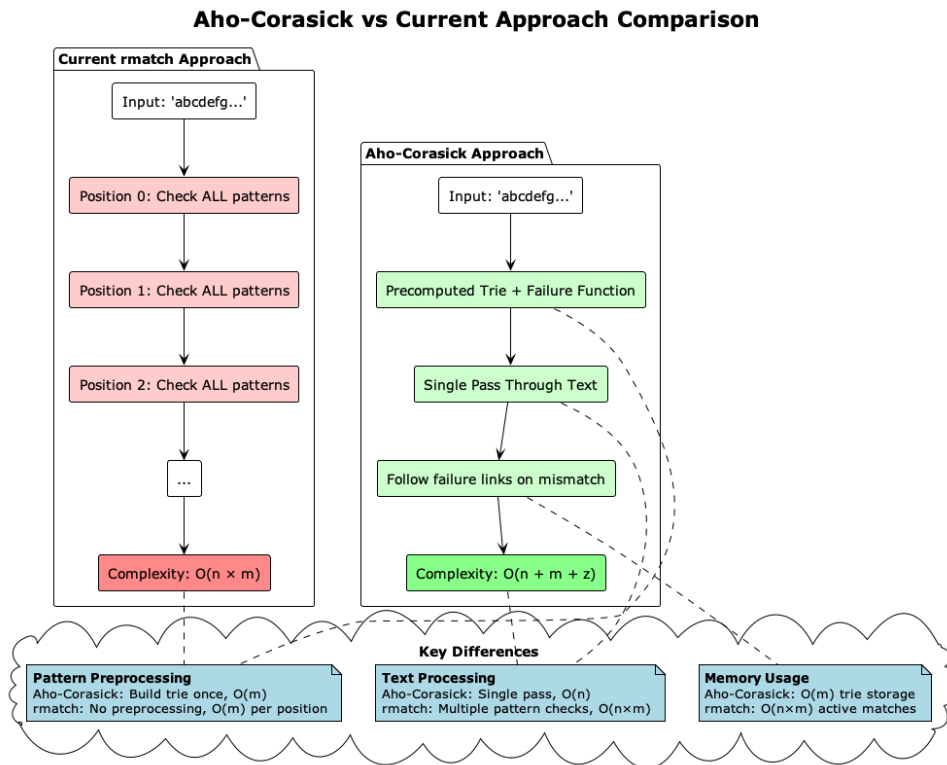


Figure 3: Aho-Corasick vs Current Approach

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

```

1:  $D_0 \leftarrow$  initial state bitvector
2: for each character  $c$  in text do
3:    $D_{i+1} \leftarrow (D_i \ll 1) \wedge T[c]$ 
4:   if  $D_{i+1} \wedge F \neq 0$  then
5:     report match
6:   end if
7: 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 $O(m \times l)$ Complexity**

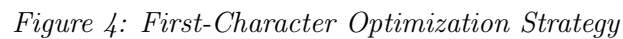
Implement first-character heuristics to eliminate the critical bottleneck:

Listing 2: Proposed first-character optimization

```

1 // Pre-compute character-to-patterns mapping
2 Map<Character, BitSet> firstCharMap = new HashMap<>();
3
4 // At each position, only consider patterns that can start with current
   char
5 BitSet candidatePatterns = firstCharMap.get(currentChar);
6 for (int patternId : candidatePatterns) {
7     // Only create matches for viable patterns
8 }

```



- Replace `SortedSet<NDFANode>` with compact `int[]` arrays
- Use lock-free data structures for multi-threading
- Implement object pooling for frequently allocated objects
- Replace string-based counters with primitive arrays

4.2.1 Hybrid Aho-Corasick Integration



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

- 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

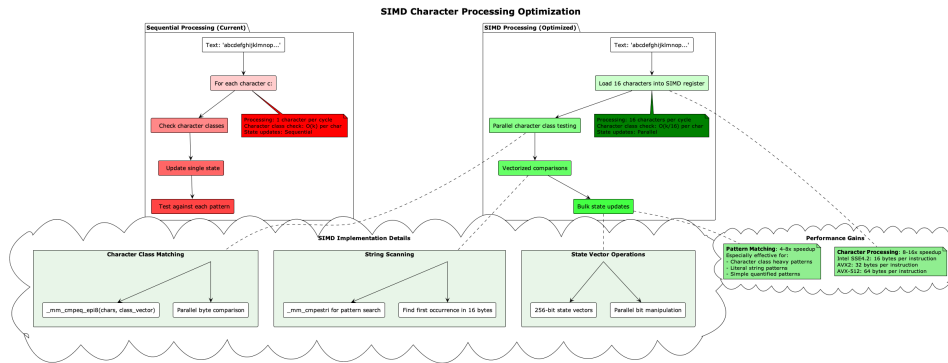


Figure 6: SIMD Character Processing

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

5 Implementation Roadmap

5.1 Development Phases

| 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

5.2 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

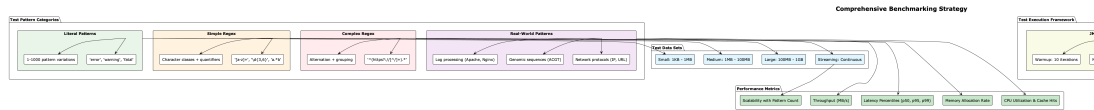


Figure 7: Comprehensive Benchmarking Strategy

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
- Google's RE2 engine (via JNI bindings)
- PCRE library performance characteristics
- Specialized multi-pattern matchers like Hyperscan

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, ...). *Swthc.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.