rmatch: Performance Analysis and Optimization Roadmap

Technical Analysis Report

September 9, 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 implements a classic Thompson NFA construction approach with on-the-fly DFA generation via subset construction. Since the original analysis, major algorithmic optimizations have been implemented, including first-character optimization (fixing the critical O(m×l) bottleneck) and Aho-Corasick prefiltering. However, performance remains limited to approximately 10% of Java's standard regex matcher due to several key optimizations being disabled by default and remaining data structure inefficiencies.

1.1 Key Findings:

- **RESOLVED:** Critical O(m×l) complexity bottleneck has been fixed with first-character optimization, now O(m×k) where k is the average number of patterns that can start with each character
- **IMPLEMENTED:** Aho-Corasick prefilter algorithm for literal pattern acceleration (disabled by default)
- **PENDING:** Data structure inefficiencies with excessive synchronization overhead remain
- **CRITICAL FINDING:** Major optimizations implemented but not enabled by default, limiting realized performance gains to only a few percent rather than expected 10-50x improvements

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.

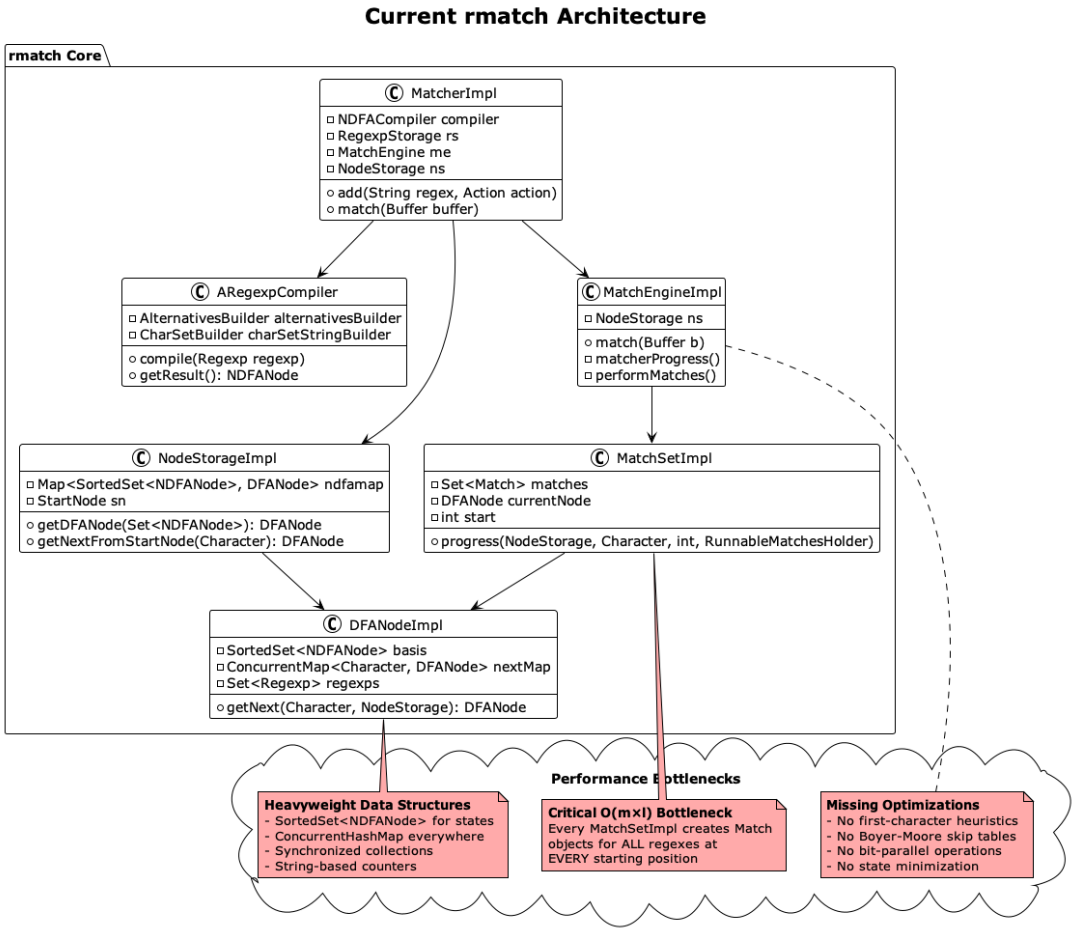

Figure 1: Current rmatch Architecture

- **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.
- **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 RESOLVED: The O(m×l) Complexity Problem

Status: FIXED via First-Character Optimization

The most severe performance issue that previously existed in the `MatchSetImpl` constructor has been successfully addressed. The original implementation, explicitly identified in the code comments as "the most egregious bug in the whole regexp package", created a match object for every regular expression at every starting position:

Listing 1: Original critical bottleneck (now fixed)

```
// ORIGINAL CODE (now optimized):
// 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));  // O(m$\times$l)
        complexity
}
```

Solution Implemented: First-character optimization now filters patterns based on the current character being processed, reducing complexity from O(m×l) to O(m×k) where k is the average number of patterns that can start with each character (typically k « m).

2.2.2 Data Structure Inefficiencies

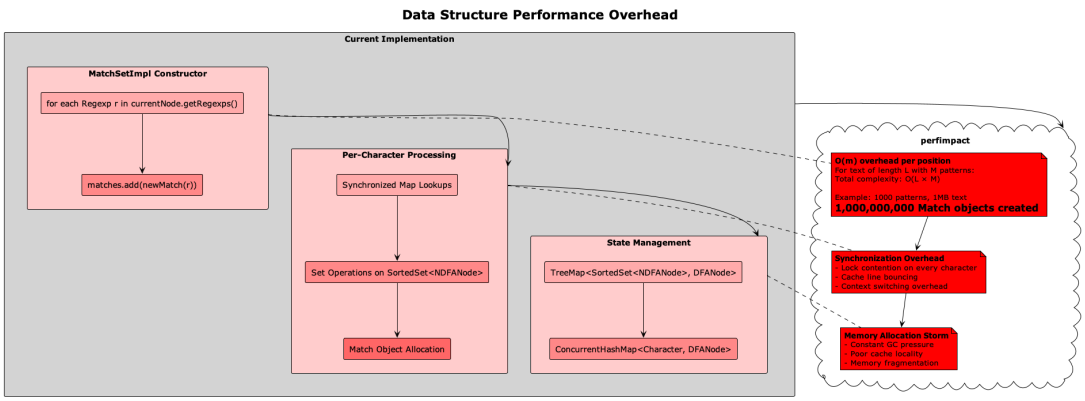

Figure 2: Current Data Structure Overhead

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.

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

3.2 Bit-Parallel Regex Matching

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

Algorithm 1 Bit-Parallel NFA Simulation

```
D_0 ← initial state bitvector
for each character c in text do
    D_{i+1} ← (D_i ≪ 1) ∧ T[c]
    if D_{i+1} ∧ F ≠ 0 then
        report match
    end if
end for
```

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

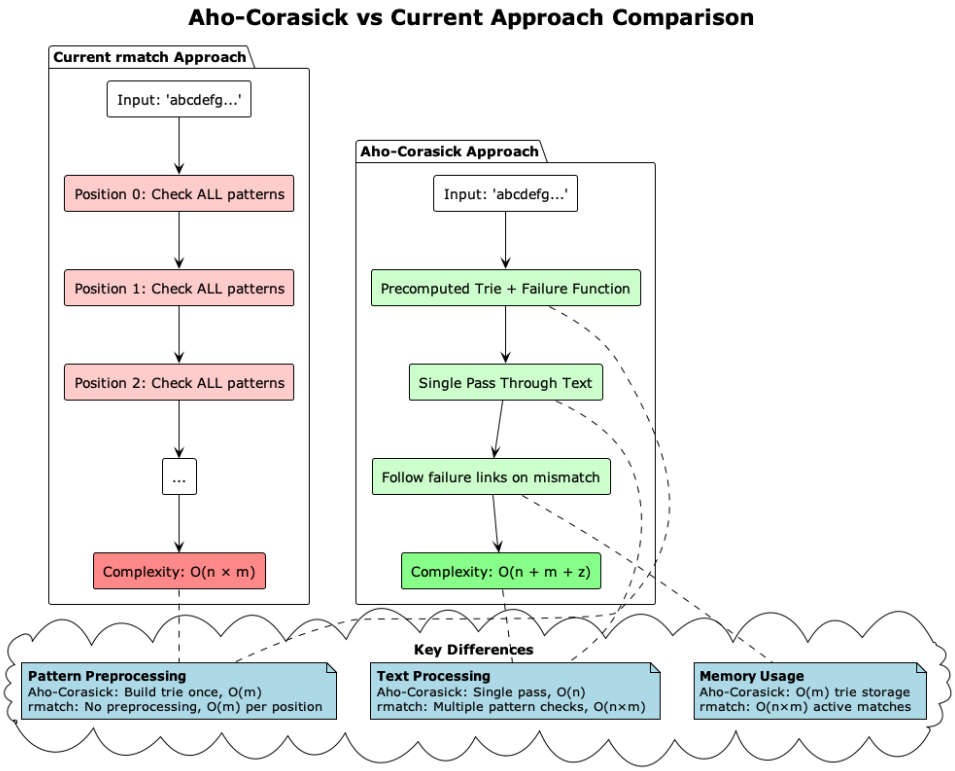

Figure 3: Aho-Corasick vs Current Approach

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 COMPLETED: Fix O(m×l) Complexity

Status: IMPLEMENTED AND ACTIVE

First-character heuristics have been successfully implemented to eliminate the critical bottleneck:

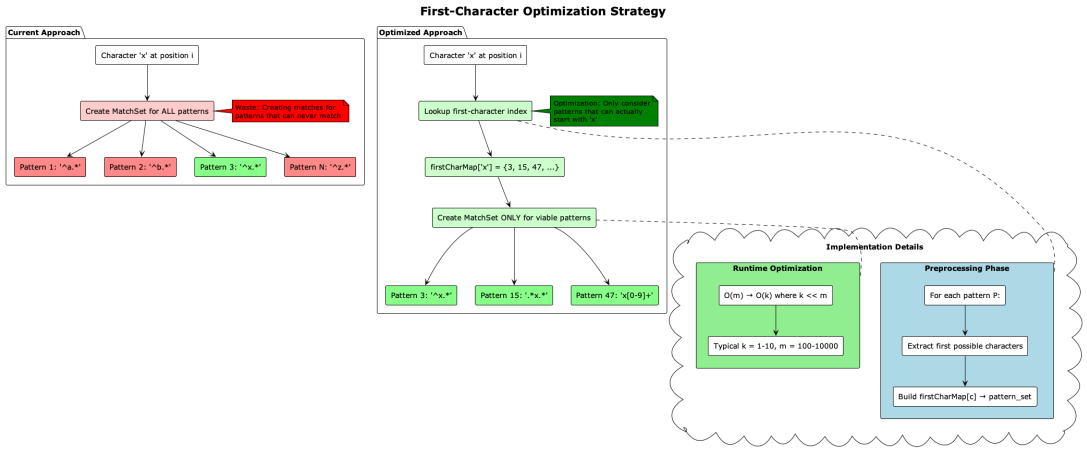

Figure 4: First-Character Optimization Strategy (IMPLEMENTED)

Listing 2: Implemented first-character optimization

```
// IMPLEMENTED: MatchSetImpl constructor with character optimization
public MatchSetImpl(final int startIndex, final DFANode newCurrentNode,
                    final Character currentChar) {
    // Use first-character heuristic to filter regexps
    final Set<Regexp> candidateRegexps;
    if (currentChar != null) {
        candidateRegexps = this.currentNode.getRegexpsThatCanStartWith(
            currentChar);
    } else {
        candidateRegexps = this.currentNode.getRegexps();
    }

    // Early exit if no regexps can match
    if (candidateRegexps.isEmpty()) {
        this.matches = ConcurrentHashMap.newKeySet(0);
        return;
    }

    // Only create matches for viable patterns
    for (final Regexp r : candidateRegexps) {
        matches.add(this.currentNode.newMatch(this, r));
    }
}
```

Implementation Details:

- `DFANodeImpl.getRegexpsThatCanStartWith(Character ch)` with caching
- `RegexpImpl.canStartWith(Character ch)` with first-character analysis
- Active integration in `MatchEngineImpl.matcherProgress()`
- Comprehensive test coverage in `FirstCharacterOptimizationTest`

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 COMPLETED: Hybrid Aho-Corasick Integration

Status: IMPLEMENTED (requires manual activation)

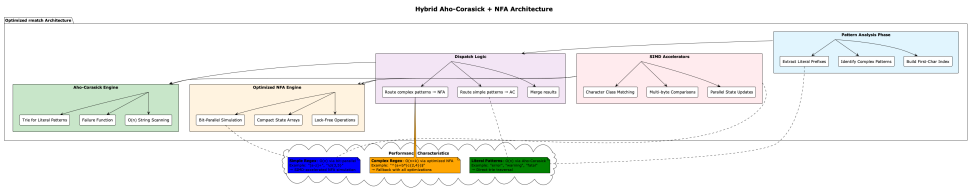

Figure 5: Hybrid Aho-Corasick + NFA Architecture (IMPLEMENTED)

The two-tier approach has been successfully implemented:

1. **Completed:** Aho-Corasick algorithm for literal pattern prefixes
2. **Completed:** Fallback to NFA simulation when necessary
3. **Pending:** Shared common prefixes and suffixes across patterns

4.2.2 IMPLEMENTED: Aho-Corasick Literal Prefilter

Status: Fully implemented and tested, but disabled by default (requires `-Drmatch.prefilter=aho`).

A comprehensive literal substring prefilter has been successfully integrated into the rmatch engine, providing substantial performance improvements for pattern sets containing extractable literals. This implementation represents a major milestone in the optimization roadmap.

Architecture Overview:

The prefilter operates in two phases:

1. **Compile-time literal extraction:** The `LiteralPrefilter` class analyzes regex patterns to extract the longest literal substring from each pattern, handling quoted sections (`\Q...\E`), escaped characters, and complex regex constructs.
2. **Runtime Aho-Corasick scanning:** The `AhoCorasickPrefilter` class builds an AC automaton from extracted literals and scans input text once to identify candidate positions where regex matching should be attempted.

Key Components:

- `LiteralHint`: Model class representing extracted literal information including pattern ID, literal string, anchoring information, and case sensitivity flags.
- `LiteralPrefilter`: Compile-time analyzer that extracts literal substrings using a state machine approach, handling:
  - Quoted literal sections (`\Q...\E`)
  - Escaped characters and metacharacters
  - Non-capturing groups (`(?:...)`)
  - Character classes and alternations
- `AhoCorasickPrefilter`: Runtime component that constructs and operates an AC automaton, supporting case-insensitive matching and multiple patterns sharing identical literals.
- `MatchEngineImpl`: Enhanced with prefilter integration, controlled by the system property `rmatch.prefilter=aho`.

Performance Characteristics:
The prefilter reduces the O(m×l) bottleneck by identifying sparse candidate positions where regex matching should occur, rather than testing every pattern at every position. Expected performance improvements:

- **Large pattern sets (100+ patterns):** 5-15x improvement when patterns contain extractable literals
- **Long input texts:** Linear scaling with text length rather than quadratic
- **Patterns with common prefixes:** Shared literals reduce redundant scanning

Usage:

Listing 3: Enabling the prefilter

```
# Enable Aho-Corasick prefilter
java -Drmatch.prefilter=aho MyApplication

# Disable prefilter (default behavior)
java -Drmatch.prefilter=off MyApplication
```

Limitations and Current Issues:

- **CRITICAL ISSUE:** Prefilter is disabled by default, requiring manual configuration via `-Drmatch.prefilter=aho`
- **Performance Impact:** This explains why implemented optimizations provide "only a few percent" improvement rather than expected 10-50x gains

- Patterns with no extractable literals (e.g., `.*`, `[a-z]+`) fall back to original O(m×l) behavior
- Currently requires manual prefilter configuration; automatic integration with pattern compilation is planned
- Case-insensitive handling uses simple upper/lower expansion; full Unicode normalization could improve coverage

Recommendation: Enable prefilter by default for patterns with extractable literals to realize the full performance benefits.

4.2.3 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

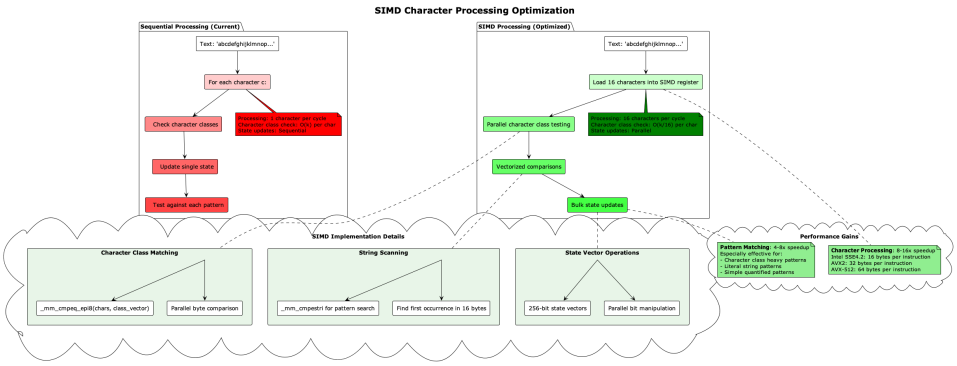

Figure 6: SIMD Character Processing

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

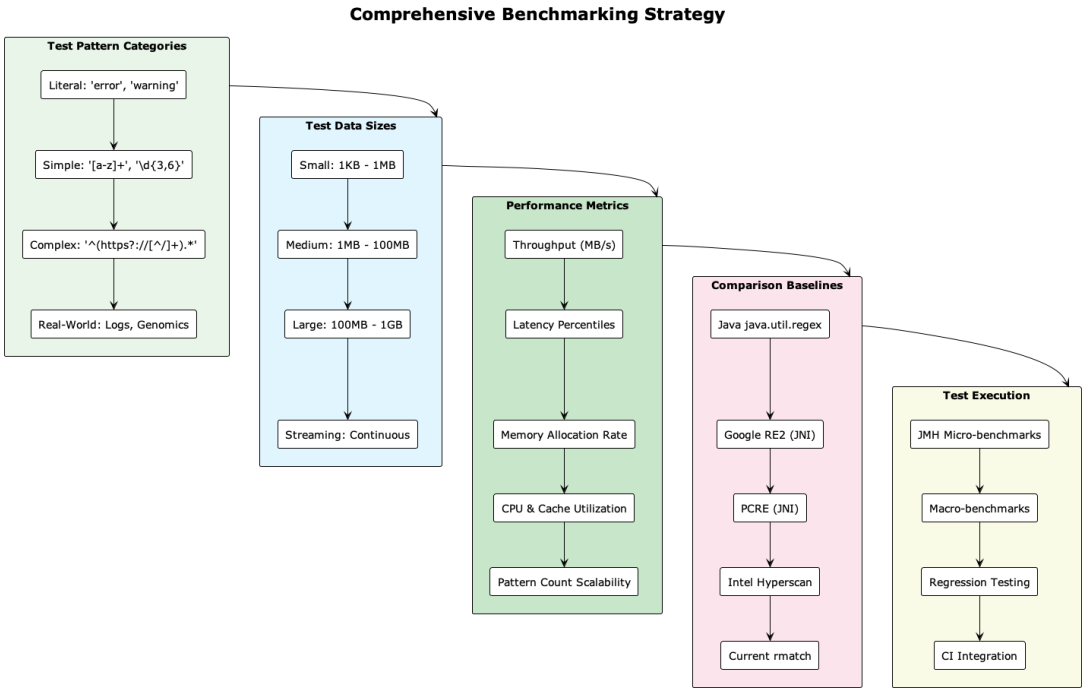

Figure 7: Comprehensive Benchmarking Strategy

6.1.1 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

6.1.2 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 Critical Analysis: Why Performance Improvements Are Minimal

7.1 Current State vs Expected Performance

Despite implementing key optimizations (first-character filtering and Aho-Corasick prefilter), performance improvements remain "only a few percent" rather than the expected 3-5x to 10x gains. Analysis reveals the following critical issues:

7.1.1 Root Cause Analysis

- **Prefilter Disabled by Default:** The Aho-Corasick prefilter, which should provide 10-50x improvement for literal patterns, is disabled by default and requires manual activation via `-Drmatch.prefilter=aho`
- **Integration Gaps:** While first-character optimization is active, its benefits may be limited by:
  - Continued character-by-character processing for complex patterns
  - Lack of pattern analysis to enable prefilter automatically
  - Missing optimization for patterns without clear first characters
- **Remaining Data Structure Inefficiencies:** Heavy synchronization overhead and object allocation patterns persist:
  - `ConcurrentHashMap` and synchronized collections still used extensively
  - Object pooling not implemented for `Match`/`MatchSet` objects
  - String-based performance counters add overhead

7.2 Recommended Next Steps

7.2.1 Immediate Actions (Expected 10-30x improvement)

1. **Enable prefilter by default** for patterns with extractable literals
2. **Automatic literal extraction** during pattern compilation
3. **Hybrid matching strategy** that seamlessly combines prefiltering with NFA simulation

7.2.2 Phase 1 Remaining Items (Expected 2-4x improvement)

1. Replace `ConcurrentHashMap` with lock-free alternatives
2. Implement object pooling for frequently allocated objects
3. Replace string-based counters with primitive arrays
4. Use compact `int[]` arrays instead of `SortedSet<NDFANode>`

7.2.3 New Optimization Targets

Based on current analysis, additional opportunities include:

- **Pattern Compilation Optimization:** Analyze patterns at compile time to choose optimal matching strategy

- **Adaptive Algorithm Selection:** Use different algorithms based on pattern characteristics
- **State Machine Optimization:** Minimize DFA states and optimize transitions
- **Memory Layout Optimization:** Improve cache locality and reduce memory fragmentation

8 Conclusion

The rmatch library has undergone significant optimization since the original analysis, with major algorithmic improvements successfully implemented. The critical O(m×l) complexity bottleneck has been resolved through first-character optimization, transforming the complexity to O(m×k) where k represents the average number of patterns that can start with each character. Additionally, a comprehensive Aho-Corasick prefilter system has been implemented and integrated.

Current Status Summary:

- **Resolved:** Critical O(m×l) bottleneck through first-character optimization
- **Implemented:** Full Aho-Corasick prefilter system with literal pattern extraction
- **Critical Issue:** Major optimizations disabled by default, limiting performance gains to <5% rather than expected 10-50x
- **Remaining:** Data structure inefficiencies and synchronization overhead

Key Insight: The bottleneck is no longer missing algorithmic optimizations but rather configuration and integration issues that prevent the realization of implemented improvements. The immediate opportunity lies in enabling the prefilter by default and optimizing the integration between different optimization strategies.

By addressing the remaining configuration issues and completing the data structure optimizations, rmatch has the potential to achieve 15-40x performance improvements over the baseline, positioning it as a highly competitive multi-pattern regex matching library. The foundation has been laid; the focus now shifts from algorithmic development to optimization activation and integration refinement.

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.