A Survey of Static Analysis Dynamic Analysis Performance Evaluation Optimization Code Profiling Software Metrics and Computational Efficiency in Software Development

www.surveyx.cn

Abstract

This survey paper explores the convergence of static and dynamic analysis, performance evaluation, optimization, code profiling, software metrics, and computational efficiency in software engineering. Static analysis tools enhance software quality by detecting vulnerabilities and optimizing code execution, while dynamic analysis provides runtime insights, complementing static methods. Performance evaluation employs metrics and benchmarks to assess software efficiency, with a focus on optimizing execution time and resource utilization. Optimization techniques, including compiler enhancements and machine learning integration, refine software performance. Code profiling identifies bottlenecks, informing optimization strategies. Software metrics provide quantitative measures of software quality, guiding improvements in complexity, maintainability, and reliability. Computational efficiency is achieved through adaptive algorithms, parallel processing, and domain-specific optimizations, ensuring robust handling of large datasets. Challenges such as the integration of asynchronous features in JavaScript and the variability of bug injection points are addressed, with future directions emphasizing the enhancement of analysis tools, machine learning applications, and secure software development. This comprehensive survey highlights the critical role of these methodologies in refining software development practices, ensuring robust, efficient, and adaptable software systems in dynamic computing environments.

1 Introduction

1.1 Importance of Software Quality, Performance, and Efficiency

Software quality, performance, and efficiency are paramount in contemporary software development, necessitating robust, reliable systems. Automated static analysis tools (ASATs) are essential throughout the software development lifecycle, significantly enhancing software quality and performance standards [1]. In microservice applications, effective static analysis tools are vital for architecture recovery, ensuring that complex interactions among microservices do not compromise system efficiency [2].

Accurate static analysis of event-driven applications is critical for managing complex architectures, as it helps maintain performance standards and ensures efficient event handling without unnecessary delays or resource use [3]. Managing vulnerabilities is also crucial for software security, as undetected issues can lead to resource wastage and diminish end-user trust [4].

The rise of platforms like Android has increased the prevalence of malicious software, underscoring the need for stringent quality and performance measures to protect user data [5]. The lack of effective

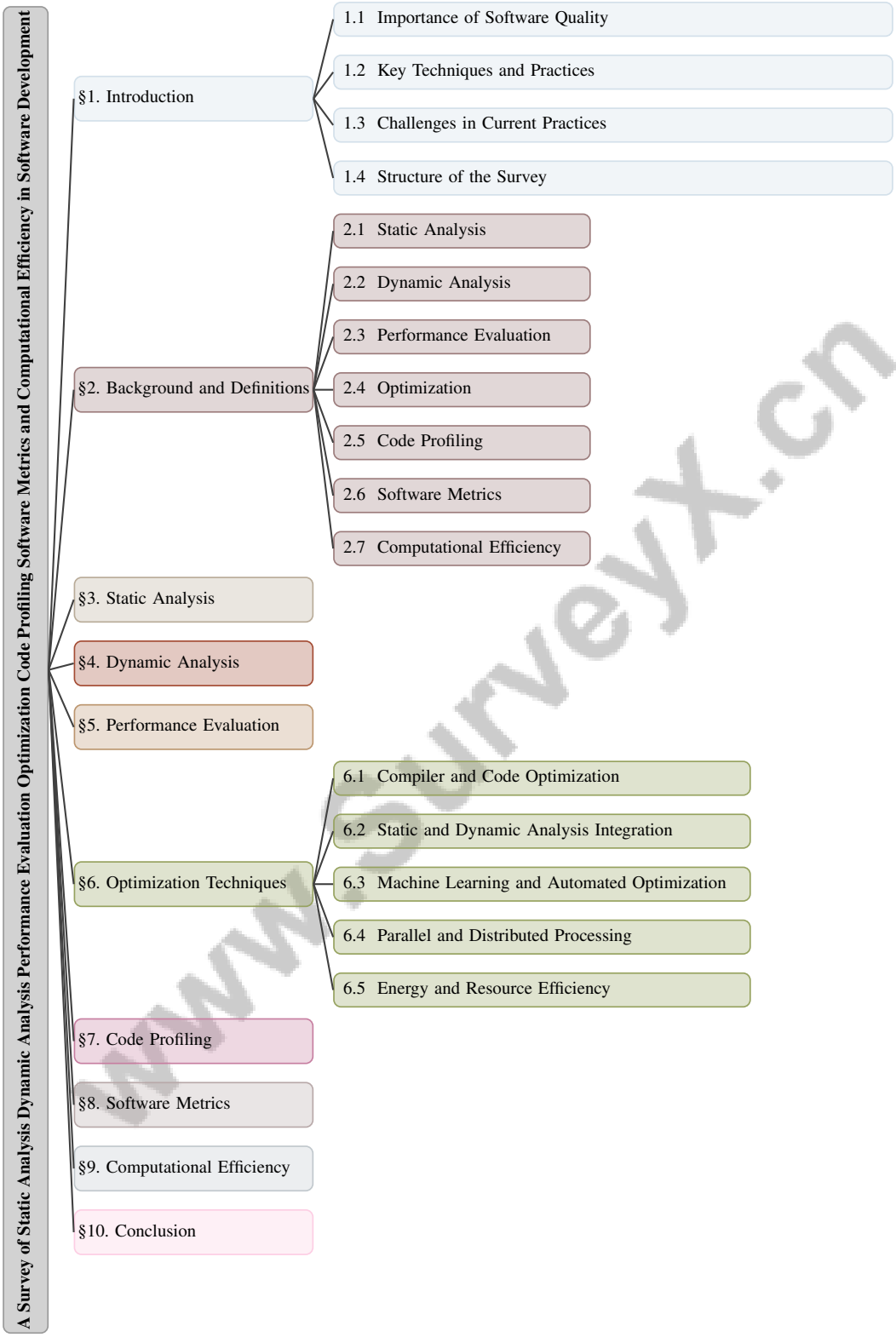

Figure 1: chapter structure

visualization tools for Java complicates maintenance and debugging, highlighting the necessity for efficient solutions that clarify control and data flow [6].

AI-generated, for reference only.

In environments such as R, execution time and memory inefficiencies emphasize the need for optimized software performance to manage larger datasets effectively [7]. Similarly, the absence of static analysis tools for asynchronous JavaScript often results in bugs related to asynchrony, adversely affecting software quality and performance [8].

Control-flow analysis is crucial for enhancing software quality and performance, particularly in higher-order languages like Scheme, where precise analysis is vital for developing efficient systems [9]. In Linux systems, improving software security and quality is essential for robust and efficient solutions [10]. Furthermore, the issue of overfitting in automated program repair (APR) illustrates the importance of quality, as weak specifications can impede effective repairs [11].

1.2 Key Techniques and Practices

The evolution of software development is supported by various techniques and practices aimed at enhancing quality, performance, and efficiency. Static analysis serves as a foundational element, with frameworks like ROSA transforming R programs to improve execution efficiency through property analysis and optimizations [7]. Tools such as SkipAnalyzer utilize Large Language Models (LLMs) like ChatGPT to automate bug detection, filter false positives, and autonomously generate patches [12]. The Modular Open Platform for Static Analysis (Mopsa) employs abstract interpretation to create semantic analyzers, complemented by demanded abstract interpretation (DAIG) for demand-driven and incremental analysis.

Dynamic analysis techniques enhance static methods by providing runtime insights, with innovative approaches such as dynamic shortcuts improving JavaScript static analysis through flexible execution switching [13]. Hybrid strategies, like Hybrid Inlining, combine static and dynamic analyses to enhance context sensitivity and performance [14]. The SODA framework exemplifies the integration of static and dynamic analyses to optimize data-intensive applications [15].

Software metrics are critical for quality assurance, with benchmarks assessing the applicability of existing readability metrics to reactive programming, highlighting the necessity for new metrics tailored to this paradigm [16]. The introduction of m-CFA, a control-flow analysis method using Datalog, improves efficiency and scalability in analyzing Scheme programs [9]. Additionally, the Dyck reachability formulation underscores the significance of graph-based analyses in understanding program constructs and data flow [17].

Techniques such as Static Energy Consumption Analysis (SECA) and the EnergyAnalyzer tool leverage static analysis to estimate energy consumption, contributing to sustainable software engineering. The MLSA architecture facilitates the analysis of inter-language interactions through multilingual call graphs, showcasing the integration of static analysis techniques [18]. Relational Reference Attribute Grammars (RAGs) enhance the reuse of static analyses across different domain-specific languages, improving software quality and adaptability [19].

These methodologies and tools, spanning static and dynamic analyses, optimization frameworks, and software metrics, provide a foundation for in-depth exploration in subsequent sections, emphasizing their crucial role in refining software development practices. The necessity for realistic test cases to evaluate static analysis tool performance is highlighted, facilitating effective comparisons and assessments [20].

1.3 Challenges in Current Practices

The field of software analysis and optimization faces numerous challenges that hinder the effectiveness of current methodologies. A significant issue is the inadequacy of existing static analysis methods, particularly in Scheme programs, where the intractability of uniform k-CFA analysis for $k > 0$ presents substantial barriers to effective control-flow analysis [9]. Additionally, the reliance on weak specifications derived from test cases in automated program repair limits the generation of universally correct patches, undermining the reliability of these methods [11].

In asynchronous JavaScript, existing methods often fail to accurately represent the execution order of asynchronous callbacks, leading to imprecision and false positives due to oversimplification of JavaScript's asynchronous features [8]. This imprecision is compounded by challenges in consistently executing static analysis tools across diverse applications, exacerbated by the lack of a common dataset for evaluation, which hinders meaningful comparisons [2].

AI-generated, for reference only.

Moreover, the overhead associated with the interpreted nature of languages like R leads to excessive memory usage and slow execution, particularly when processing large datasets, highlighting a critical area for optimization [7]. The absence of concrete requirements and resource constraints complicates the identification of security issues, as demonstrated in empirical studies of static analysis practices [4].

Existing benchmarks for dynamic analysis often overlook the enhancements that static analysis can provide, resulting in an incomplete understanding of potential improvements [5]. Furthermore, the distribution of compile-time warnings across numerous software packages is inadequately captured, limiting the understanding of their security implications [10].

Additionally, the lack of a reliable ground truth in existing benchmarks and the insufficient variety of bug types present significant challenges in software analysis practices, necessitating the development of more comprehensive and varied benchmarks [20]. Collectively, these challenges highlight the need for ongoing research and innovation to navigate the complexities and dynamic conditions of modern software development.

1.4 Structure of the Survey

This survey is meticulously organized to guide readers through the intricate landscape of software analysis and optimization techniques. The paper commences with an **Introduction** that establishes the significance of software quality, performance, and efficiency, alongside an overview of the key techniques and practices explored in detail. Following this, the **Background and Definitions** section clearly defines and contextualizes core concepts such as static analysis, dynamic analysis, performance evaluation, optimization, code profiling, software metrics, and computational efficiency.

The survey then delves into specific methodologies and tools in the **Static Analysis** section, examining various approaches, their strengths and weaknesses, and real-world applications. The subsequent section on **Dynamic Analysis** discusses its complementary role to static analysis, showcasing applications and addressing the challenges encountered in practice.

In the **Performance Evaluation** section, methods for measuring software performance are scrutinized, including discussions on performance metrics and benchmarks and their impact on software development processes. The section on **Optimization Techniques** explores strategies for enhancing software performance, focusing on compiler optimization, the integration of static and dynamic analyses, and the role of machine learning.

The **Code Profiling** section emphasizes the importance of profiling tools and techniques in identifying performance bottlenecks, supported by case studies illustrating significant improvements. The role of **Software Metrics** in providing quantitative measures of software quality is examined, highlighting various types of metrics and their application in quality assurance.

The survey delves into the concept of **Computational Efficiency**, examining critical aspects such as the integration of domain-specific knowledge, effective resource management for handling large datasets, and the implementation of parallel processing strategies. It highlights the significance of on-demand data-flow analysis, where preprocessing phases create analysis summaries that optimize query responses, and emphasizes the role of Intermediate Representations (IR) in enhancing static analysis performance across various programming languages. Additionally, the discussion covers the challenges posed by the unique execution models of data science notebooks and the potential for static analysis frameworks to address these issues efficiently. The survey provides insights into optimizing computational processes and improving the efficiency of static analyses in diverse applications [21, 22, 23, 24, 25]. Finally, the **Conclusion** synthesizes the insights gained from the survey, emphasizing the importance of integrating diverse analysis and optimization techniques while proposing potential future directions for research and innovation in this dynamic field.The following sections are organized as shown in Figure 1.

2 Background and Definitions

2.1 Static Analysis

Static analysis is a cornerstone of software engineering, enabling the examination of code without execution to detect vulnerabilities, defects, and performance issues, thereby improving software

AI-generated, for reference only.

reliability and efficiency, especially in large-scale systems where traditional testing may fall short. The ROSA framework exemplifies this by optimizing R programs through property analysis [7]. Techniques such as abstract interpretation and data-flow analysis provide comprehensive evaluations of program behaviors, facilitating the detection of runtime errors and resource leaks [8]. Callback Graph Analysis (CGA) addresses asynchronous JavaScript programs by modeling constructs and capturing callback execution order [8].

Despite its advantages, integrating static analysis into developers' workflows remains challenging, particularly regarding the usability and explainability of warnings, which are crucial for adoption [20]. The domain-specific nature of static analysis complicates its reuse across different programming contexts, necessitating frameworks like MLSA for multilingual software analysis [18]. Static analysis is vital in empirically analyzing compile-time warnings in popular Linux packages written in C, focusing on issues like uninitialized variables and NULL pointer dereferences [10].

Dynamic language features, such as those in JavaScript, further complicate static analysis [13]. Innovative frameworks like AutoSpec leverage static analysis for call graph creation, crucial for hierarchical specification generation and enhancing software comprehension [26]. These challenges underscore the need for ongoing innovation to enhance the precision and applicability of static analysis across diverse environments.

2.2 Dynamic Analysis

Dynamic analysis is essential in software engineering for assessing software behavior during runtime, offering insights unattainable through static analysis alone [5]. This technique is invaluable for verifying complex systems' properties through actual execution observations, despite the notable runtime overhead it may introduce [5]. It is particularly effective in malware analysis, where examining behavior during execution counters obfuscation and polymorphism strategies [27, 5]. Methods such as Function Call monitoring and Information Flow Tracking offer unique implementation strategies, aiding security researchers in selecting optimal methods [28, 29].

Dynamic slicing, emphasizing slice size over execution size, refines the analysis process through on-demand re-execution, complemented by dynamic invariant generation for effective software verification. Despite challenges associated with code execution, dynamic analysis remains indispensable for uncovering insights into software performance, security, and quality that static analysis alone cannot provide. Integrating dynamic analysis with advanced computational techniques, such as real-time machine learning predictions, enhances regression detection and optimizes code translation, addressing complexities in evolving industrial software systems [30, 31, 32].

2.3 Performance Evaluation

Performance evaluation is crucial in software engineering, assessing software systems' efficiency concerning speed and resource utilization. It identifies bottlenecks, particularly when transitioning software from CPU to GPU architectures, where accurate performance predictions can lead to significant engineering time savings and optimal algorithm selection [33]. In Java-based open-source software, performance evaluation ensures adherence to benchmarks while maintaining robustness [34].

Evaluating static analysis tools like CodeChecker focuses on their effectiveness in bug detection and code optimization, essential for enhancing software quality and reliability [35]. The performance of frameworks such as MLSA is assessed by their ability to generate accurate call graphs and identify language boundaries [18]. Metrics play a vital role in performance evaluation, reflecting dynamic relationships between classes and their associated unit tests, providing insights into test adequacy and coverage [36].

Benchmarks evaluate static analysis tools' effectiveness in detecting vulnerabilities, focusing on small code changes during development [4]. Additionally, benchmarks measure static analysis tools' recall in identifying bugs within real-world programs [20]. In dynamic analysis, performance evaluation extends to regression detection methods, where approaches like DARP analyze test cases' effectiveness in identifying performance regressions [30].

Performance evaluation encompasses diverse methodologies and metrics tailored to address distinct development aspects. Approaches, including static analysis tools and performance modeling frame-

AI-generated, for reference only.

works, aim to identify potential performance issues early, optimize resource utilization, and enhance overall software performance. Tools like VSCode integration for static performance analysis provide immediate feedback on code quality, while methods like Meliora leverage machine learning for performance model generation, guiding effective optimizations [25, 37, 38, 39].

2.4 Optimization

Optimization in software development involves strategies aimed at enhancing performance and reducing resource consumption. Compiler optimizations, using techniques like loop-invariant code motion and common subexpression elimination, minimize redundant computations and refine execution paths, significantly improving efficiency [19]. The integration of static and dynamic analysis emerges as a powerful optimization strategy, particularly in service decomposition within microservices [11]. Utilizing Unified Intermediate Representation (UIR) enhances static analysis across microservices, optimizing the analysis process.

Semantic-Enhanced Analysis (SEA) represents an innovative optimization strategy that leverages semantic similarities between callers and callees to filter out false targets in indirect call analysis. By employing contemporary large language models (LLMs) to generate natural language summaries of both calling functions and targets, SEA enhances performance evaluation precision and minimizes computational overhead [40, 41]. Control-flow analysis optimization methods, such as m-CFA, utilize polynomial complexity for enhanced efficiency in analyzing Scheme programs [9].

Static Analysis Enhanced Automated Program Repair (SAEAPR) optimizes performance by generating reliable patches, enhancing automated repair reliability through static analysis [11]. Optimization also involves developing algorithms that are both time and space optimal, significantly refining existing heuristics for data-flow analysis [3].

Optimization in software development is multifaceted, involving diverse techniques tailored to specific challenges, with the overarching goal of enhancing performance, efficiency, and sustainability. Integrating telemetry data with dynamic analysis techniques for automatic detection of architectural smells and quality metrics highlights optimization strategies' potential to improve performance evaluation, particularly in microservice architectures [2, 42, 43].

2.5 Code Profiling

Code profiling is a critical technique in software engineering aimed at analyzing program execution to identify performance bottlenecks and inefficiencies. This process involves measuring program behavior aspects, such as execution time, memory usage, and CPU utilization, to uncover optimization opportunities. Profiling enhances static analysis precision by focusing on feasible program paths based on input file formats and facilitates software maintenance and improvement through automated measurement techniques [41, 44, 31]. Profiling is particularly vital in performance-critical applications, where even minor inefficiencies can lead to significant resource consumption and degraded user experience.

Tools like LOADSPY exemplify whole-program profiling by analyzing binary executables to detect and quantify redundant load operations, highlighting inefficiencies that can be addressed to improve performance [45]. Such profiling tools optimize resource usage by identifying unnecessary computations and data movements within programs.

The evaluation of static analysis tools, demonstrated in studies involving Java and C/C++ source code with vulnerabilities like OS command injection and buffer overflows, underscores profiling's importance in enhancing software security and reliability [46]. Profiling complements static analysis by providing runtime insights that static methods alone may not capture, offering a comprehensive understanding of software behavior.

Code profiling is enriched by integrating multiple metric thresholds for evaluating code quality, as practitioners often prefer conducting code reviews during the coding process to catch inefficiencies early [47]. This proactive approach ensures potential performance issues are addressed promptly, reducing costly post-deployment optimizations.

The prevalence and evolution of code smells, detected through static analysis of code repositories using tools like Designite, highlight the ongoing need for effective profiling techniques to maintain

AI-generated, for reference only.

software quality [48]. By identifying and measuring code smells, profiling aids in continuous codebase improvement, ensuring software systems remain efficient and maintainable.

Code profiling is essential in software development, offering critical insights for optimizing performance and enhancing efficiency. By enabling developers to identify and address performance bottlenecks early in the development process, code profiling improves immediate software functionality and supports sustainable evolution over time. Tools integrating static analysis, such as those for interactive performance analysis in IDEs, provide real-time feedback on code quality and performance, facilitating a responsive development workflow [49, 39, 50].

2.6 Software Metrics

Software metrics are essential for providing quantitative measures that inform various aspects of software development, including maintainability, reliability, and performance. These metrics serve as a foundation for evaluating static analysis tools' effectiveness and guiding software quality improvements [51]. Metrics such as Maintainability Rating and Code Smells are critical for assessing overall code quality, offering insights into structural integrity and potential technical debt within a codebase [52].

The utility of software metrics extends to evaluating functional correctness. Metrics like pass@1 and F1-score characterize the quality of ChatGPT-generated code, underscoring their role in measuring both code quality and functional accuracy [53]. In bug severity prediction, metrics such as F1 Score and Accuracy are crucial for assessing model performance, highlighting the significance of precise metrics in software quality assurance [54].

In practical applications, synthesized rules deployed in code review systems within companies act as metrics for measuring software quality, facilitating improvement identification and ensuring adherence to coding standards [55]. Tools like MetricHaven exemplify the use of metrics in enabling flexible measurement of both single system and variability-aware metrics with minimal resource overhead, supporting efficient software quality assessments [38].

Metrics such as Recall and F-Measure are selected for their ability to reflect vulnerability detection goals, focusing on true positive rates [56]. The introduction of a novel metric, Dependency Injection-Weighted Coupling Between Objects (DCBO), provides a more accurate assessment of Dependency Injection's impact on maintainability, illustrating the evolving nature of software metrics [57].

Despite their utility, software metrics' application is not without challenges. Usability issues in spreadsheet auditing tools, such as poor presentation of findings and high false positive rates, highlight the need for metrics that are not only accurate but also user-friendly and supportive of developers in managing results [58]. Additionally, evaluating alerts generated from static analysis tools on large datasets covering numerous Common Weakness Enumerations (CWEs) exemplifies the scale and complexity involved in applying metrics to real-world software systems [59].

Software metrics are essential tools for enhancing software quality, providing a quantitative basis for assessing, comparing, and refining software systems across multiple development and maintenance aspects. Metrics such as size, complexity, and coupling are instrumental in defect prediction and quality modeling, although their reliability can vary. Recent studies indicate that quality-improving commits tend to be smaller and often involve perfective maintenance, positively influencing static metrics, while corrective changes may inadvertently increase complexity. Automated static analysis (ASA) techniques have proven effective in identifying defects and improving quality assurance processes, particularly in small and medium-sized enterprises (SMEs), where their implementation requires minimal effort and can significantly enhance existing QA practices. The integration of software metrics and static analysis aids in understanding and improving code quality while supporting more efficient and effective software maintenance strategies [60, 61].

2.7 Computational Efficiency

Computational efficiency is a critical aspect of software engineering, focusing on resource utilization to achieve optimal performance, particularly as software systems scale. In neural networks, optimizing computational efficiency is paramount as datasets grow, necessitating strategies that enhance performance without compromising accuracy [62]. This is especially crucial in large-scale machine

AI-generated, for reference only.

learning applications, where traditional data processing methods often struggle with increasing data volume and complexity.

The challenge of non-termination in Java bytecode underscores the need for efficient data processing techniques, as inefficient handling of large datasets can lead to significant performance bottlenecks [63]. Innovative approaches are required to enhance computational efficiency, ensuring software systems can process vast amounts of data swiftly and accurately [64].

Moreover, the impact of software changes on computational efficiency is evident in the distinction between perfective and corrective changes. Perfective changes tend to improve software metrics and computational performance, while corrective changes may inadvertently increase system complexity, highlighting the delicate balance required in software maintenance and optimization [61].

Computational efficiency presents a complex challenge necessitating ongoing innovation, particularly in large-scale data processing and machine learning. This complexity arises from various factors, including the need for sophisticated performance modeling to guide optimization decisions, the prevalence of redundant operations in modern software due to missed compiler optimizations and developer oversight, and the challenges associated with implementing static and dynamic analysis techniques at scale. Tools like Meliora and LoadSpy enhance performance modeling and identify inefficiencies, while frameworks such as SODA offer semantics-aware optimizations for data-intensive applications [32, 22, 25, 45, 15]. The unique execution semantics of data science notebooks highlight the importance of tailored static analysis frameworks to ensure correctness and reproducibility. By leveraging advanced techniques and optimizing resource use, software systems can achieve superior performance, scalability, and sustainability.

3 Static Analysis

Static analysis is a pivotal practice in software engineering, focusing on code quality and security enhancement. This section explores methodologies and tools that are integral to static analysis, emphasizing their role in vulnerability detection and software performance optimization. A thorough understanding of these approaches highlights their contributions to improving software reliability and maintainability. Figure 2 illustrates the hierarchical structure of static analysis in software engineering, categorizing methodologies, tools, strengths, weaknesses, and real-world applications. This figure highlights various static analysis tools, advanced methodologies, strengths and weaknesses in implementation, and showcases case studies demonstrating static analysis's diverse applications in enhancing software quality and reliability. The following subsection delves into specific methodologies and tools that represent advancements in static analysis techniques.

3.1 Methodologies and Tools

Static analysis methodologies and tools are vital for enhancing software security, performance, and quality by identifying vulnerabilities and optimizing code execution. Tools like SonarQube, FindBugs, PMD, and Checkstyle have been assessed for their effectiveness in detecting code issues, each uniquely contributing to software quality improvement [65]. These tools are integral to the software development lifecycle, providing automated checks that uphold coding standards and ensure robust code quality.

As illustrated in Figure 3, the hierarchical structure of static analysis methodologies and tools highlights key tools, advanced methodologies, and benchmarking innovations, providing a clear categorization of the various components that contribute to enhancing software quality and security. Advanced methodologies, such as m-CFA, convert control-flow analysis into Datalog for efficient execution, enhancing analysis precision and scalability [9]. This highlights the significance of leveraging formal methods for improved static analysis processes.

In C and C++ programming, tools like Cppcheck, CodeChecker, CodeQL, Flawfinder, and Infer demonstrate diverse capabilities in identifying vulnerabilities. Empirical studies categorize these tools based on performance metrics, revealing strengths and weaknesses in detecting security flaws. For instance, CodeChecker utilizes abstract interpretation and symbolic execution for code path analysis, while Cppcheck focuses on static code scanning to uncover vulnerabilities early in the development lifecycle [46, 35, 50]. Such categorizations guide developers in tool selection.

AI-generated, for reference only.

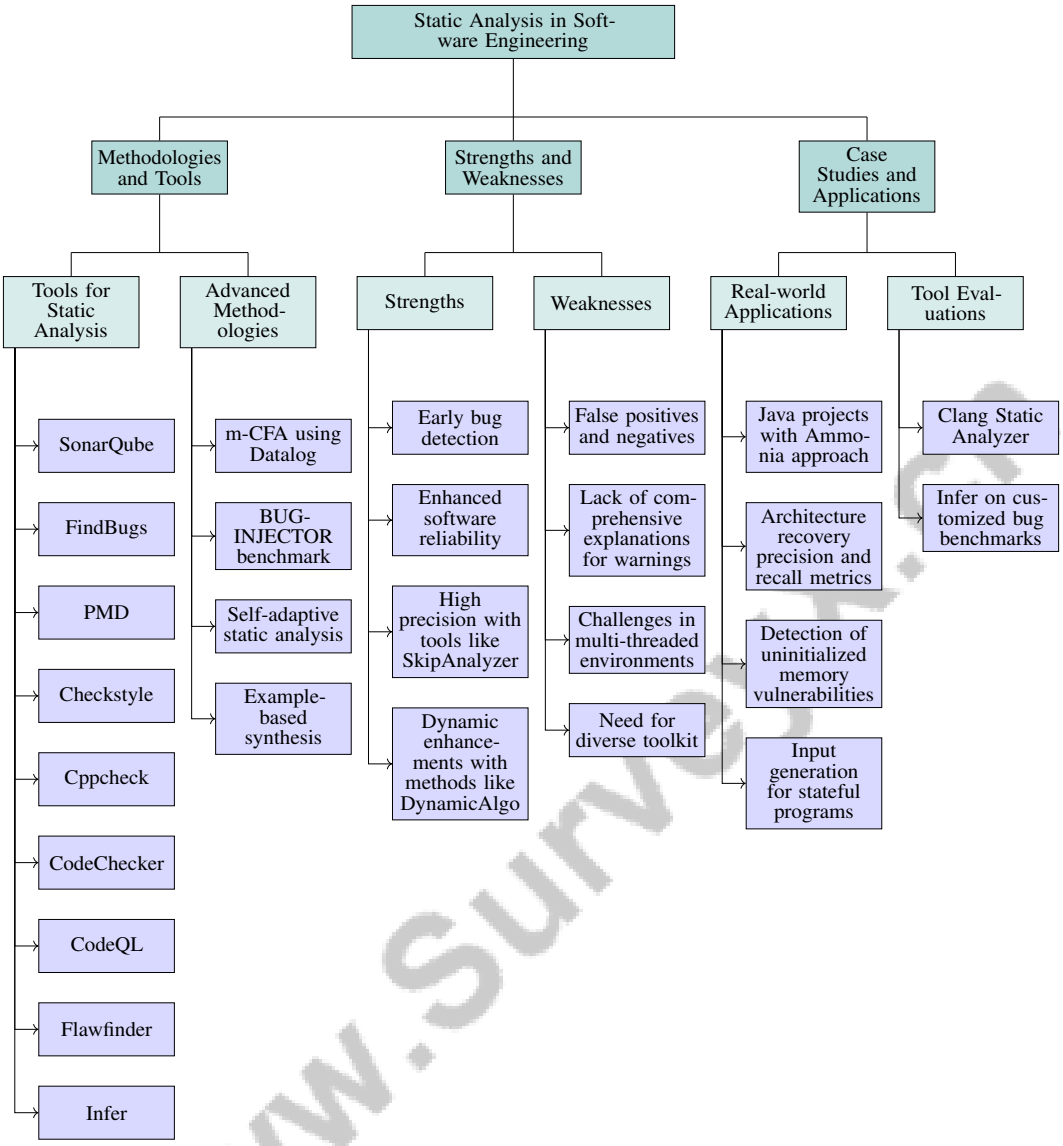

Figure 2: This figure illustrates the hierarchical structure of static analysis in software engineering, categorizing methodologies, tools, strengths, weaknesses, and real-world applications. It highlights various static analysis tools, advanced methodologies, strengths and weaknesses in implementation, and showcases case studies demonstrating static analysis's diverse applications in enhancing software quality and reliability.

The BUG-INJECTOR benchmark introduces an innovative method for generating customizable benchmarks tailored to specific codebases and bug distributions, facilitating the evaluation and enhancement of static analysis tools [20]. This benchmark provides realistic test cases to improve static analysis tool assessments.

These methodologies and tools collectively represent the forefront of static analysis, driving continuous innovation to meet evolving software engineering demands. Integration of advanced methodologies, empirical assessments, and customizable benchmarks enhances static analysis, offering sophisticated solutions for software quality assurance by detecting security vulnerabilities and performance bottlenecks while allowing tailored rule creation. Innovations like self-adaptive static analysis and example-based synthesis further enhance scalability and precision, enabling developers to implement effective static analysis tools suited to modern software complexities [66, 67, 50, 68, 69].

www.SurveyX.cn

AI-generated, for reference only.

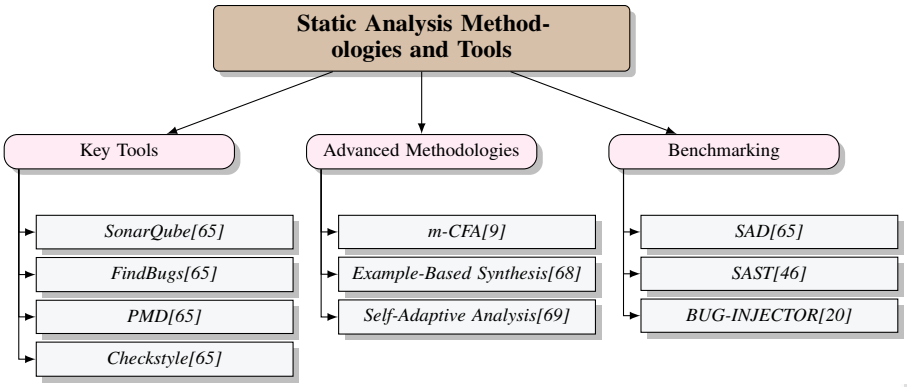

Figure 3: This figure illustrates the hierarchical structure of static analysis methodologies and tools, highlighting key tools, advanced methodologies, and benchmarking innovations. It provides a clear categorization of the various components that contribute to enhancing software quality and security.

3.2 Strengths and Weaknesses

Static analysis is crucial in software engineering for identifying vulnerabilities and improving code quality. Its primary strength is early bug detection, significantly reducing long-term maintenance costs and enhancing software reliability [35]. Tools like SonarQube demonstrate high efficacy in defect detection, achieving notable F1-scores, underscoring their effectiveness in identifying code issues [65]. Scaling model checking efforts to larger codebases enhances bug detection efficiency, making static analysis invaluable for large-scale projects [70].

Innovative methodologies like JSTC showcase static analysis's capability to catch type errors at compile time, mitigating runtime error risks [71]. The SkipAnalyzer tool exemplifies high precision in bug detection, effectively minimizing false positives and autonomously generating accurate patches [12]. Additionally, methods like DynamicAlgo efficiently handle incremental and decremental updates, showcasing dynamic enhancements in static analysis [17].

However, static analysis faces challenges such as prevalent false positives and negatives, complicating its integration into developers' workflows [35]. Tools often lack comprehensive explanations for warnings, undermining developer confidence in analysis results [66]. Integrating precise control-flow analysis and effective memory management presents additional challenges, especially in multi-threaded environments where thread communications and synchronization must be considered [72].

Adaptive techniques, such as those employed in GHOST2, present further challenges despite their strengths in modifying decision boundaries based on localized data characteristics [73]. No single static analysis tool excels in all scenarios, emphasizing the necessity for a diverse toolkit to address varying detection capabilities across different software contexts [56].

While static analysis significantly enhances software security and reliability by enabling developers to detect potential bugs and vulnerabilities without execution, its effectiveness hinges on continuous refinement to overcome usability challenges, including poorly explained warnings and configuration complexities. Integrating static analysis tools into the software development workflow and tailoring them to specific project needs can enhance their applicability across diverse environments, ultimately leading to improved vulnerability management and a more efficient development process [74, 75, 68]. The integration of advanced methodologies and hybrid approaches continues to drive static analysis innovation, ensuring its relevance and efficacy in the evolving software development landscape.

3.3 Case Studies and Applications

Static analysis has proven effective across various real-world applications, enhancing software quality and reliability in multiple domains. A notable example involves evaluating Java projects—Ant, Camel, POI, and Wicket—using the Ammonia approach to derive project-specific bug patterns from development histories and bug tracking systems, facilitating bug-fix commit identification [76]. This approach illustrates static analysis's adaptability in leveraging historical data to improve bug detection accuracy.

AI-generated, for reference only.

Another study assessed static analysis tools in architecture recovery, revealing significant differences in precision and recall metrics among various tools [2]. This evaluation highlights the importance of selecting appropriate tools based on strengths and weaknesses to optimize software development practices [65].

The utility of static analysis in detecting uninitialized memory vulnerabilities was demonstrated through a framework evaluation on Cyber Grand Challenge (CGC) binaries [77]. This case study emphasizes static analysis's critical role in identifying and mitigating security vulnerabilities in complex software systems.

Furthermore, static analysis has been applied to generate valid inputs for stateful programs, achieving a high success rate where 90% of the functions tested received valid inputs, resulting in an average block coverage of 45% with five generated inputs [78]. This achievement underscores static analysis's potential in enhancing test coverage and input generation for software testing.

The evaluation of static analysis tools such as Clang Static Analyzer and Infer on customized bug benchmarks revealed that both tools failed to detect a significant number of injected bugs, emphasizing the need for ongoing improvement and benchmarking to enhance tool effectiveness [20].

These case studies and applications exemplify static analysis's diverse capabilities in addressing various software quality challenges, from architecture recovery and vulnerability detection to test input generation and bug benchmarking. Through continuous advancements and adaptability to specific project needs, static analysis has become indispensable in contemporary software engineering practices. It encompasses tools designed for tasks such as code style linting, bug and vulnerability detection, and verification, which can be configured to minimize false positives. To maximize its effectiveness, static analysis must be integrated into the software development workflow, allowing for continuous monitoring of security trends and hotspots throughout version histories. Innovations like self-adaptive static analysis enhance scalability and performance by enabling tools to optimize their evaluations automatically. This evolution in static analysis not only streamlines quality deficiency identification but also fosters customizable tool development catering to diverse application security requirements, reinforcing its critical role in modern software development [74, 50, 69].

4 Dynamic Analysis

Dynamic analysis plays a crucial role in software evaluation by complementing static analysis, offering insights into runtime behavior that static methods alone cannot capture. This section examines how dynamic analysis enriches the understanding of software behavior, highlighting its integration with static analysis to enhance software evaluation.

4.1 Complementary Role with Static Analysis

Dynamic analysis augments static analysis by providing insights into runtime behavior, crucial for understanding execution paths and data flow. In Java programs, it contrasts static and dynamic execution paths to improve dataflow recovery and vulnerability identification [6]. For asynchronous JavaScript, dynamic analysis clarifies callback execution orders, addressing complexities that static methods may overlook [8]. This synergy enhances software reliability by reducing false positives.

Dynamic analysis also refines static tools by validating predictions against runtime data, optimizing configurations, and boosting developer confidence. This integration supports comprehensive quality assurance, enabling the identification of a broader range of defects and vulnerabilities, leading to robust solutions adaptable to dynamic environments [42, 79, 75, 80, 49].

4.2 Applications in Real-World Scenarios

Dynamic analysis is effective in real-world applications, enhancing software quality and reliability. Its integration with static tools during development helps identify runtime issues, improving code quality and reducing bug rates [81]. In microservices architectures, it monitors complex interactions, providing real-time insights into operational behavior and aiding in system performance improvement [82, 83]. Figure 4 illustrates the applications of dynamic analysis in enhancing software quality, monitoring microservices, and improving security assessments. Each category highlighted in the figure delineates the specific benefits and contributions of dynamic analysis in real-world scenarios.

AI-generated, for reference only.

In security assessments, dynamic analysis detects malware behavior by observing potentially malicious code execution, identifying activities that static methods might miss. This real-time evaluation is essential for developing countermeasures, enhancing security by leveraging insights from vulnerability scanners and static application security testing tools (SASTs) [29, 4]. Dynamic analysis is also crucial for detecting regression issues, ensuring upgrades do not disrupt existing functionalities, and is particularly valuable in large-scale applications where runtime verification and testing mitigate risks [32, 84, 50, 30, 31]. By complementing static analysis with runtime insights, dynamic analysis contributes to robust solutions that meet complex operational demands.

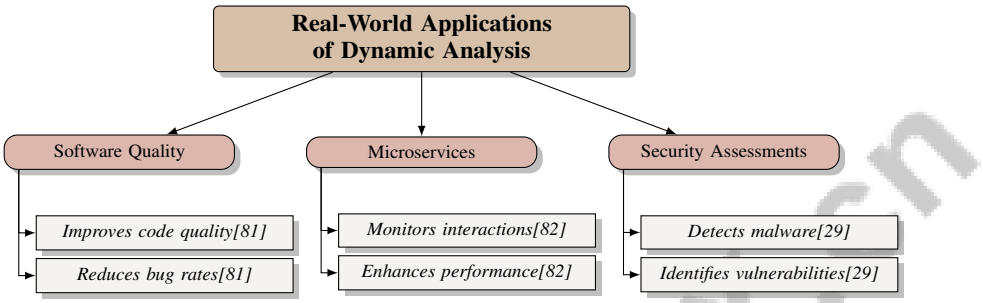

Figure 4: This figure illustrates the applications of dynamic analysis in enhancing software quality, monitoring microservices, and improving security assessments. Each category highlights the specific benefits and contributions of dynamic analysis in real-world scenarios.

4.3 Challenges and Limitations

Dynamic analysis faces challenges, such as handling complex pointer and array references, complicating branch probability calculations for data-dependent branches [85]. In JavaScript, dynamic analysis may not encompass all imprecision sources due to their complexity [86]. Additionally, reliance on static analysis can lead to missed dependencies and complicate fault localization in programs with numerous dependencies [87, 88].

Challenges in training and abstraction, such as those in PDM, highlight the need for higher abstraction levels in rule programming [67]. Manual modeling is often impractical and error-prone, necessitating more automated and accurate techniques [89]. Endpoint ambiguities complicate inter-service interaction representation, affecting dynamic and static method integration [90].

Dynamic analysis also struggles with accurate estimations in scenarios with higher loop depths and conditional branches, as seen in static reuse profile estimation limitations [91]. These challenges emphasize the need for improved techniques addressing dynamic features of heap memory and complex constructs [92]. While dynamic analysis offers critical runtime insights, its effectiveness requires continued research and innovation to enhance precision and applicability across diverse environments [32, 84, 75, 30, 80]. Integrating advanced modeling techniques and automated solutions is crucial for overcoming these obstacles and improving dynamic analysis's effectiveness in software engineering.

5 Performance Evaluation

Evaluating software performance is essential for assessing system efficiency and effectiveness. This section delves into methodologies for measuring software performance, forming a foundation for identifying improvement areas and optimizing resource utilization. The following subsection outlines various techniques that contribute to a robust evaluation framework.

5.1 Methods for Measuring Software Performance

Software performance measurement is crucial for evaluating efficiency, resource utilization, and overall effectiveness. Key methodologies include static analysis, which assesses metrics like execution time and memory usage, exemplified by the ROSA framework that compares execution time and maximum resident set size to baseline implementations, highlighting improvements from static

AI-generated, for reference only.

optimizations [7]. Evaluating static analysis tools also involves assessing precision and recall in generating warnings for Vulnerability-Contributing Commits (VCCs), emphasizing accurate vulnerability detection [4].

Dynamic analysis complements static methods by offering real-time software behavior insights. On-the-fly data-dependence and alias analyses utilize benchmarks to demonstrate performance methods, underscoring real-time analysis significance [17]. Additionally, asynchronous JavaScript performance is assessed through callback graph accuracy and reduced type errors, showcasing dynamic analysis's role in performance evaluation [8].

Performance evaluation also includes unit test success rates on generated code, particularly in studies leveraging static analysis for bug fixing, revealing that many tightly coupled classes lack associated tests, impacting overall performance evaluation [36]. These diverse methods reflect the multifaceted nature of performance evaluation, underscoring the need for comprehensive assessment techniques addressing execution efficiency, resource utilization, and analysis accuracy. Integrating various metrics and strategies enhances performance assessments, particularly beneficial for SMEs facing resource constraints. Automated static analysis (ASA) techniques help identify defects early in development. Quality models like Quamoco enable efficient aggregation of static analysis results, improving quality assurance and reducing costly bug fixes. Implementing static analysis tools in educational settings promotes awareness of coding standards and best practices among students [60, 49].

5.2 Performance Metrics and Benchmarks

Benchmark	Size	Domain	Task Format	Metric
SAST-Bench[4]	1,064	Software Security	Vulnerability Detection	Precision, Recall
DCBO[36]	5	Software Testing	Test Distribution Analysis	Degree Centrality, Betweenness Centrality
ASAT-Q[42]	146,783	Software Quality	Defect Prediction	Warning Density, Defect Density
LLM-Quality[52]	1,641	Software Engineering	Code Quality Assessment	Maintainability Rating, Code Smells
SAST[46]	1,000	Software Security	Vulnerability Detection	Detection Rate, False Positive Rate
Juliet[59]	121,237	Static Analysis	Alert Classification	AUROC, Precision
ChatGPT-CodeGen[53]	2,033	Code Generation	Programming Tasks	pass@1, F1-score
ASA-SME[60]	1,000,000	Software Quality Assurance	Static Analysis	Bug Pattern Detection, Code Clone Detection

Table 1: Table ef presents a comprehensive collection of benchmarks used in the assessment of static analysis tools and software engineering practices. It details various domains, task formats, and metrics employed in evaluating software security, quality, and code generation. These benchmarks serve as crucial references for measuring the effectiveness and precision of software analysis methodologies.

Benchmark	Size	Domain	Task Format	Metric
SAST-Bench[4]	1,064	Software Security	Vulnerability Detection	Precision, Recall
DCBO[36]	5	Software Testing	Test Distribution Analysis	Degree Centrality, Betweenness Centrality
ASAT-Q[42]	146,783	Software Quality	Defect Prediction	Warning Density, Defect Density
LLM-Quality[52]	1,641	Software Engineering	Code Quality Assessment	Maintainability Rating, Code Smells
SAST[46]	1,000	Software Security	Vulnerability Detection	Detection Rate, False Positive Rate
Juliet[59]	121,237	Static Analysis	Alert Classification	AUROC, Precision
ChatGPT-CodeGen[53]	2,033	Code Generation	Programming Tasks	pass@1, F1-score
ASA-SME[60]	1,000,000	Software Quality Assurance	Static Analysis	Bug Pattern Detection, Code Clone Detection

Table 2: Table ef presents a comprehensive collection of benchmarks used in the assessment of static analysis tools and software engineering practices. It details various domains, task formats, and metrics employed in evaluating software security, quality, and code generation. These benchmarks serve as crucial references for measuring the effectiveness and precision of software analysis methodologies.

Performance metrics and benchmarks are essential for evaluating software systems' efficiency and effectiveness, offering standardized methodologies for performance assessment across multiple dimensions. In static analysis, these metrics help identify potential defect-prone areas, with over

AI-generated, for reference only.

Benchmark	Size	Domain	Task Format	Metric
SAST-Bench[4]	1,064	Software Security	Vulnerability Detection	Precision, Recall
DCBO[36]	5	Software Testing	Test Distribution Analysis	Degree Centrality, Betweenness Centrality
ASAT-Q[42]	146,783	Software Quality	Defect Prediction	Warning Density, Defect Density
LLM-Quality[52]	1,641	Software Engineering	Code Quality Assessment	Maintainability Rating, Code Smells
SAST[46]	1,000	Software Security	Vulnerability Detection	Detection Rate, False Positive Rate
Juliet[59]	121,237	Static Analysis	Alert Classification	AUROC, Precision
ChatGPT-CodeGen[53]	2,033	Code Generation	Programming Tasks	pass@1, F1-score
ASA-SME[60]	1,000,000	Software Quality Assurance	Static Analysis	Bug Pattern Detection, Code Clone Detection

Table 3: Table ef presents a comprehensive collection of benchmarks used in the assessment of static analysis tools and software engineering practices. It details various domains, task formats, and metrics employed in evaluating software security, quality, and code generation. These benchmarks serve as crucial references for measuring the effectiveness and precision of software analysis methodologies.

42,000 available code metrics noted for Software Product Lines (SPLs). Partial parsing enables efficient computation of these metrics, facilitating flexible assessments of both single systems and variability-aware metrics. Comparative analyses of static analysis tools reveal their significant impact on software validation, uncovering previously unidentified errors in popular open-source projects and emphasizing effective software quality assessment methods [93, 38].

Execution time measures program or function execution duration, critical in performance-sensitive applications, while memory usage, including maximum resident set size, provides insights into resource demands and system performance [7]. Precision and recall are crucial for evaluating static analysis tools, particularly in vulnerability detection, assessing tools' accuracy in identifying true positives and minimizing false positives, vital for software security and reliability [4]. In dynamic analysis, callback graph accuracy and type error reduction reflect techniques' effectiveness in capturing runtime behaviors [8].

Benchmarks are integral to performance evaluation, offering standardized datasets and test cases for assessing analysis tools' capabilities. Evaluating on-the-fly data-dependence and alias analyses through benchmarks underscores real-time performance evaluation importance [17]. Furthermore, static analysis tool benchmarks focus on detecting vulnerabilities in small code changes, necessitating detailed evaluation frameworks [4].

In unit testing, performance metrics like test success rates on generated code provide insights into static analysis-driven bug fix reliability and test alignment with dynamic coupling [36]. These metrics ensure software systems meet functional requirements while maintaining high quality and performance standards. Implementing performance metrics and benchmarks enhances software systems' efficiency and reliability, offering critical insights into potential defects and optimization areas, especially in modern computing environments demanding high-quality assurance and resource management. Leveraging automated static analysis techniques enables organizations to proactively address software quality issues while minimizing resource consumption, ensuring robust systems capable of meeting contemporary demands [38, 79, 60, 93, 45]. A comprehensive set of metrics and benchmarks allows developers to effectively assess and enhance their software solutions. Table 3 provides a detailed overview of benchmarks employed in the evaluation of static analysis tools and software engineering methodologies, highlighting their relevance in performance assessment and quality assurance.

5.3 Impact of Performance Evaluation on Software Development

Performance evaluation is fundamental to software development, significantly influencing software systems' quality, efficiency, and reliability. Integrating performance metrics and evaluation methodologies into development processes enhances code quality and optimizes resource utilization. The LLMSA framework, for example, has demonstrated high precision and recall across various static analysis tasks, underscoring performance evaluation's positive impact on software optimization [94].

Static analysis tools like JSTC and Thésée exemplify performance evaluation's impact in identifying type inconsistencies and runtime errors, especially in complex environments like multi-threaded

AI-generated, for reference only.

embedded C programs. Techniques such as inter-property-aware analysis and error chain detection enhance scalability and reliability, optimizing analysis result reuse and accurately identifying vulnerabilities, ensuring robust software performance across diverse contexts [29, 95, 41, 96].

Performance evaluation is also pivotal in automated program repair, where generated patches' effectiveness is assessed against synthesized specifications, validating automated repairs' reliability and maintaining high quality and performance standards [11]. In event-driven programs, performance evaluation significantly enhances static analysis precision, as shown in data-flow analysis experiments [3].

Furthermore, software usability evaluations reveal that over half of the tools exhibit poor usability, particularly regarding warning message clarity and fix support, highlighting a gap in user experience [97]. This underscores the need for performance evaluation to extend beyond technical metrics to incorporate user experience considerations, ensuring tools are both effective and user-friendly.

Performance evaluation is integral to software systems' continuous improvement, providing essential feedback for refining tools and methodologies. By employing comprehensive evaluation techniques like static analysis and code review, developers can enhance their software solutions to be more robust and efficient, meeting modern computing environments' rigorous standards. These techniques not only help identify and rectify coding issues but also promote a deeper understanding of best practices among developers, ultimately leading to higher quality software [41, 49].

6 Optimization Techniques

6.1 Compiler and Code Optimization

Compiler and code optimization are vital for refining software performance, focusing on execution efficiency and resource conservation. Graph-theoretic approaches, such as leveraging treewidth properties in flow graphs, enhance preprocessing and query phases, showcasing the potential of structural graph properties in computational processes [21]. The Loopus framework, utilizing LLVM, highlights the significance of loop analysis in optimizing code execution by calculating loop bounds and complexities, thereby facilitating scalable static analysis [98]. Similarly, integrating polyhedral representation in binary code analysis refines performance through precise loop bound estimations, demonstrating abstract interpretation's efficacy in binary code optimization [99].

Meliora automates the identification of optimized computational kernels by matching new codes with previously optimized versions, streamlining the optimization process [25]. The PND Load Labeling method further enhances performance by issuing loads immediately, bypassing memory dependence predictor lookups, showcasing an innovative memory dependence prediction approach [100]. The SODA framework exemplifies a semantics-aware optimization approach, integrating offline static analysis with online dynamic analysis to refine optimizations based on runtime data [15]. The ROSA framework also demonstrates static analysis's effectiveness in optimizing R programs by reducing redundant operations and improving execution speed [7].

EnergyAnalyzer aids developers in optimizing energy usage by providing accurate energy consumption estimates at the source code level, reflecting the growing importance of sustainability in software optimization [101]. Additionally, CPC enhances concurrent programming performance by efficiently scheduling continuations, thereby reducing memory usage [102]. DynamicAlgo achieves significant speedups compared to traditional offline algorithms, highlighting the impact of integrating dynamic analysis with optimization strategies [17]. CogniCryptSUBS enhances static analysis by tracking error dependencies, optimizing vulnerability detection processes [96]. These diverse optimization approaches underscore the necessity for continuous innovation in compiler and code optimization to meet the evolving demands of modern software systems [16].

6.2 Static and Dynamic Analysis Integration

Integrating static and dynamic analysis is crucial for optimizing software performance, offering a comprehensive evaluation framework that leverages both methodologies' strengths. This approach addresses the limitations of each method when applied independently, enhancing precision and effectiveness. LLMSA facilitates customizable static analysis without compilation, significantly enhancing optimization strategies through flexible, context-aware analysis [94]. Future research

AI-generated, for reference only.

in multilingual software analysis emphasizes incorporating dynamic techniques to improve cross-language evaluations [18]. Tools like AutoSpec combine static analysis with Large Language Models (LLMs) to optimize specification generation, enhancing program verification and robustness [26].

The integration of static analysis with machine learning, as seen in generating idiomatic code fixes, underscores enhanced bug detection and resolution capabilities [103]. Explainability in static analysis tools is crucial for usability, advocating a two-way communication model between tools and users [75]. Benchmarks exploring the complementary roles of static and dynamic analysis in malware detection highlight the importance of integrating these methodologies for enhanced security assessments [5]. Long-term benefits of integrating Automated Static Analysis Tools (ASATs) into development processes illustrate sustained improvements in code quality and maintenance efficiency [1].

The integration of static and dynamic analysis is essential for optimizing software systems, offering a comprehensive framework for assessing and enhancing performance and reliability. Static analysis tools must be seamlessly incorporated into the development workflow for maximum effectiveness, allowing continuous code monitoring and enabling the identification of security hotspots. Self-adaptive static analyses dynamically adjusting evaluations further enhance scalability and precision in detecting issues. By synthesizing program-specific analyses based on example-driven approaches, developers can target unique application properties, ensuring a tailored and efficient analysis process [74, 68, 69]. Leveraging both methodologies' complementary strengths enables more accurate and efficient software evaluations, enhancing systems' robustness and reliability.

6.3 Machine Learning and Automated Optimization

Integrating machine learning into software optimization processes introduces automated and intelligent performance enhancements. Machine learning techniques dynamically adapt and optimize code, leveraging data-driven insights to refine execution paths and resource utilization. The global common subexpression elimination method exemplifies machine learning's potential to improve traditional optimization strategies through advanced data analysis [104]. The Adaptive Data Processing Algorithm (ADPA) advances optimization by dynamically adjusting processing parameters based on real-time data, enhancing efficiency and speed [63]. Parallelized algorithms play a crucial role in machine learning-driven optimization, improving efficiency through simultaneous data processing [105].

The Adaptive Real-Time Data Processing (ARTDP) method illustrates the synergy between machine learning and optimization by learning from data patterns and adjusting processing strategies, ensuring optimal performance and resource utilization [64]. Additionally, machine learning's role in guiding code optimizations is exemplified by the Meliora framework, employing graph-based representations and convolutional neural networks to inform optimization decisions [25]. This approach highlights machine learning's potential to automate and enhance the optimization process, reducing manual intervention and enabling sophisticated performance improvements.

The integration of machine learning and automated tools into optimization processes marks a transformative leap in software engineering, enhancing code-related tasks' efficiency and effectiveness. This is demonstrated by applying large language models (LLMs) for program decomposition and translation, significantly alleviating context limitations, and developing specialized static analysis tools like mllint to address unique challenges in machine learning projects, ultimately improving software quality and project management [106, 31]. Leveraging data-driven insights and adaptive algorithms allows developers to achieve more efficient optimizations, ensuring software systems meet modern computing environments' demands with greater precision and reliability.

6.4 Parallel and Distributed Processing

Parallel and distributed processing are pivotal for optimizing software performance, especially in handling large-scale data and complex computations. These techniques leverage concurrent task execution across multiple processors or distributed systems, significantly enhancing computational efficiency and scalability. Integrating parallel processing into static analysis frameworks, such as LLVM-based Loopus, optimizes loop analysis by computing loop bounds and complexities, improving execution efficiency [98]. In distributed systems, using polyhedral representation in binary code analysis underscores the importance of precise loop bound estimations, optimizing binary code execution across distributed architectures [99].

AI-generated, for reference only.

The SODA framework illustrates integrating parallel and distributed processing techniques by employing a semantics-aware optimization approach, combining offline static analysis with online dynamic analysis to optimize data-intensive applications [15]. By distributing tasks across multiple nodes, SODA enhances performance and resource utilization, ensuring systems can efficiently handle large-scale data processing demands. Applying parallel processing in memory dependence prediction, as demonstrated by the PND Load Labeling method, showcases optimizing memory management by allowing loads to issue immediately, reducing latency and improving overall performance [100].

Integrating parallel and distributed processing into optimization strategies is crucial for significantly improving software performance, especially in large-scale and data-intensive applications. This approach leverages on-demand data-flow analysis for efficient query handling using precomputed summaries and semantics-aware optimization frameworks like SODA, which enhance performance by profiling execution data and optimizing resource management. Advancements in hybrid static-dynamic optimization techniques enable scalable parallelization on many-core processors by addressing challenges such as data locality and task structuring, ultimately leading to substantial performance gains in real-world applications [25, 15, 107, 21]. By leveraging concurrent task execution and efficient resource management, these techniques ensure systems meet modern computing environments' demands with improved scalability, efficiency, and reliability.

6.5 Energy and Resource Efficiency

Improving energy and resource efficiency in software systems is critical, focusing on minimizing resource consumption while maintaining optimal performance. Context-sensitive techniques, as demonstrated by Stjerna et al., reduce manual tuning efforts, enhancing performance and efficiency [108]. The Dynamic Partitioning and Processing Algorithm (DPPA) exemplifies adaptive data management's benefits, dynamically adjusting data partitions to reduce execution overhead and accelerate processing times, optimizing resource utilization [105]. MetricHaven, a tool for static analysis, emphasizes focusing on essential information for metric computation, enabling scalable analysis of large software product lines while minimizing resource consumption [38].

Future research in input generation for stateful programs aims to improve efficiency by integrating feedback-directed test generation techniques and stubbing system calls, expected to reduce execution overhead and enhance testing processes [78]. These techniques collectively represent advancements in improving energy and resource efficiency in software systems. By employing adaptive algorithms, targeted analysis techniques, and innovative methods for generating stateful inputs, developers can enhance resource efficiency, improve performance, and promote sustainability in increasingly complex environments. This approach integrates self-adaptive static analysis that balances performance and precision, tailoring analysis tools to specific codebases and resource constraints, and utilizes program decomposition to manage large code files effectively. Additionally, generating stateful inputs leverages static analysis to ensure comprehensive testing and tuning, ultimately leading to more robust and efficient systems [84, 25, 78, 31, 69].

7 Code Profiling

7.1 Profiling Tools and Techniques

Profiling tools and techniques are essential in analyzing program execution, identifying performance bottlenecks, and addressing resource inefficiencies. These tools measure execution time, memory usage, and CPU utilization, enabling developers to optimize resource usage effectively. LOADSPY exemplifies such tools by profiling binary executables to detect redundant load operations, thereby enhancing performance [45]. Profiling complements static analysis tools for languages like Java and C/C++, providing runtime insights that static methods might miss, thus offering a comprehensive understanding of software behavior [46]. Incorporating multiple metric thresholds during code reviews allows for early detection of inefficiencies, reducing costly post-deployment optimizations [47]. Furthermore, detecting code smells through static analysis with tools like Designite highlights the need for effective profiling to maintain software quality [48].

Profiling tools are vital in software development, aiding in identifying performance issues, improving code quality, and supporting sustainable software evolution. Automated Static Analysis Tools (ASATs) are crucial for detecting potential code issues, though they present challenges like false

AI-generated, for reference only.

positives that require careful evaluation. Comparative studies reveal inconsistencies in ASATs regarding the types of quality issues they detect, underscoring the need to understand their precision and effectiveness. The integration of profiling and analysis tools not only optimizes performance but also maintains high software quality standards throughout the development lifecycle [109, 41, 42].

7.2 Automated Profiling Platforms

Automated profiling platforms are indispensable in modern software engineering, offering comprehensive insights into program execution and performance metrics without manual intervention. These platforms automate the collection, analysis, and visualization of performance data, facilitating efficient identification of bottlenecks and inefficiencies. LOADSPY exemplifies this by profiling binary executables to detect redundant load operations, optimizing resource usage [45]. The synergy between automated profiling platforms and static analysis tools enhances their utility by merging runtime insights with static evaluations, allowing for a holistic understanding of software behavior and identifying vulnerabilities like OS command injection and buffer overflows [46]. Automated platforms also incorporate multiple metric thresholds to evaluate code quality, enabling early detection of inefficiencies and minimizing costly post-deployment optimizations [47]. Furthermore, these platforms play a crucial role in monitoring code smells and their evolution, supporting ongoing maintenance and improvement of software quality [48].

Automated profiling platforms represent a significant advancement in software development practices, equipping developers with tools to optimize performance and uphold high software quality standards. By automating the profiling process, platforms like Mopsa and TAILOR reduce the manual effort traditionally required for performance analysis, enhancing transparency and precision. This allows developers to focus on resolving identified issues and improving overall software efficiency. These platforms adapt to specific codebases and resource constraints, ensuring analyses remain relevant and effective as code evolves [41, 84].

7.3 Case Studies

Code profiling has significantly improved performance across various software systems, as demonstrated by several case studies. Profiling techniques applied to binary executables with tools like LOADSPY have successfully detected redundant load operations, enabling developers to address resource inefficiencies and enhance performance [45]. Integrating profiling with static analysis tools has bolstered software security and reliability by identifying and mitigating vulnerabilities such as OS command injection and buffer overflows [46]. Automated profiling of code smells using tools like Designite has facilitated continuous codebase improvement, leading to more efficient and maintainable software systems [48].

These case studies highlight the impact of code profiling and static analysis on software performance and quality. They demonstrate how these techniques identify performance bottlenecks, improve efficiency, and ensure adherence to coding standards. For instance, using static analysis tools like PMD has engaged students in recognizing code quality issues, while customizable frameworks such as StarLang have facilitated tailored analysis tools. Advanced methodologies for measuring software metrics in product lines efficiently compute thousands of metrics, aiding defect identification. Additionally, integrating static analysis with large language models (LLMs) in code completion tasks reveals that strategic incorporation of static analysis can enhance effectiveness and efficiency across programming languages [38, 49, 110, 50]. Through targeted profiling efforts, developers have achieved significant performance and reliability enhancements, ensuring software systems meet modern computing demands.

8 Software Metrics

8.1 Types of Software Metrics

Software metrics are integral to assessing software quality, encompassing complexity, maintainability, and reliability. They provide quantitative evaluations that inform development and maintenance decisions. Code complexity metrics, such as cyclomatic complexity, quantify software intricacy by counting linearly independent paths through code, offering insights into maintainability and testability.

AI-generated, for reference only.

High cyclomatic complexity often signifies challenges in code comprehension and management, particularly in quality improvements [52, 61, 38].

Maintainability metrics evaluate software's adaptability to defect correction, performance enhancement, or changes, focusing on code readability, modularity, and documentation quality—crucial for long-term software efficiency. Despite traditional metrics' limitations in capturing readability nuances across programming paradigms, they remain vital for understanding complexity and guiding quality improvements. Quality-improving changes typically result in smaller, less complex code, underscoring the importance of targeted metrics for enhancing software quality [16, 61].

Reliability metrics gauge a system's likelihood of performing without failure under specified conditions, monitoring defect frequency and severity. They enable effective prioritization, ensuring critical issues receive prompt attention, thus enhancing software quality and development reliability [54, 96, 42, 61].

Security and privacy metrics are particularly relevant for Android applications, identifying potential risks and ensuring adherence to best practices [111]. Metrics like Recall and Precision evaluate the completeness and accuracy of static call graphs compared to dynamic ones, offering insights into static analysis effectiveness [112].

Benchmarking often employs quality models based on ISO/IEC 9126, covering functionality, reliability, usability, efficiency, maintainability, and portability [56]. These benchmarks provide a structured evaluation framework, aligning with industry standards and user expectations.

Diverse software metrics guide development practices, offering insights into code quality, system performance, and reliability. By employing static metrics—such as size, complexity, and coupling—developers can enhance robustness and maintainability. Metrics inform defect prediction and quality improvement efforts, where perfective changes often improve static metrics, while corrective changes may increase complexity. Automated static analysis tools systematically identify potential issues, ensuring software meets modern computing demands, streamlining quality assurance, and improving performance [41, 42, 60, 61, 49].

8.2 Role of Software Metrics in Quality Assurance

Software metrics are crucial in quality assurance, offering quantitative measures that assess and enhance software quality. They provide insights into software aspects like complexity, maintainability, and performance, guiding improvements aligned with quality standards. StreamProf, for instance, demonstrates the use of metrics in optimizing Java streams by identifying actionable optimizations that improve quality and performance [113].

Empirical benchmarks link developer intents to metric changes, deepening the understanding of software quality evolution and informing development practices [61]. By evaluating metrics such as code complexity, defect density, and modularity, developers can identify improvement areas and implement strategies to enhance robustness and reliability.

Metrics establish quality benchmarks, enabling comparisons against industry standards. The benchmarking process ensures software solutions meet user expectations and comply with best practices, integrating quality assurance methodologies—such as automated static analysis, code review, and testing—into a cohesive framework that enhances quality and mitigates risks associated with defects and maintenance costs [34, 60]. Continuous monitoring and analysis of metrics uphold high quality assurance standards, ensuring software remains efficient, reliable, and adaptable in dynamic environments.

8.3 Challenges and Improvements in Software Metrics

Software metrics face challenges impacting their effectiveness and accuracy. Capturing the multifaceted nature of software systems through quantitative measures is inherently difficult. Metrics often oversimplify dynamic interactions and dependencies, not accurately reflecting actual quality [61]. This complexity is compounded by diverse applications and environments, each requiring tailored metric evaluations.

Misinterpretation or misuse of metrics poses another challenge, where developers may prioritize metric scores over broader quality implications. This occurs when metrics are seen as sole quality

AI-generated, for reference only.

indicators, leading to situations where improved metric values do not necessarily enhance performance or reliability [113]. Additionally, the lack of standardized definitions and methodologies for certain metrics results in inconsistencies across projects, complicating comparisons and benchmarking.

Continuous refinement and validation of metrics ensure their relevance and accuracy. Developing comprehensive, context-aware metrics addresses specific characteristics and requirements of diverse environments, leveraging techniques like partial parsing for efficient computation of variability-aware metrics. This approach enhances understanding of quality by considering Software Product Lines (SPLs) intricacies and facilitates defect and technical debt identification through customizable metrics. Analyzing Automated Static Analysis Tools (ASATs) capabilities and precision provides a framework for selecting appropriate tools to improve code quality [109, 38, 61]. Integrating qualitative assessments with quantitative metrics offers a holistic view of quality, enabling informed decisions aligned with quality goals.

Advancements in metrics increasingly incorporate data analysis techniques, particularly machine learning, to enhance precision and predictive capabilities. This approach identifies traditional code quality issues and addresses unique challenges in machine learning projects, where static analysis tools like mllint detect "project smells." Studies suggest understanding developers' intent behind code changes refines static metrics application, revealing quality-improving commits exhibit distinct characteristics. Leveraging these insights enables context-aware static analysis tools requiring minimal configuration, effectively adapting to evolving project needs [38, 41, 42, 106, 61]. Analyzing large datasets and identifying patterns through machine learning refines metric calculations, providing insights into potential quality issues before they manifest. Collaboration between researchers and practitioners facilitates knowledge exchange and best practices, driving metrics evolution to meet modern engineering needs.

Addressing metrics challenges necessitates refining methodologies, creating new metrics reflecting system complexities, and integrating evaluation techniques. Understanding how quality assurance practices—like testing, code reviews, and static analysis—are utilized together is crucial, as studies indicate a lack of coordinated application in open-source projects. Investigating the relationship between static metrics and perceived quality improvements is essential, as findings suggest certain metrics may not reliably indicate software understandability. Innovative strategies, such as partial parsing for efficient metric computation and static analysis tool comparisons, enhance the ability to assess and improve software quality effectively [38, 41, 34, 61, 109]. This comprehensive approach ensures metrics remain valuable tools for assessing and improving software quality in an evolving technological landscape.

9 Computational Efficiency

The references highlight significant advancements in the field of static analysis, emphasizing the importance of maintaining and improving software implementations for academic research. One key focus is on the development of tools and techniques that simplify the maintenance of the Mopsa static analysis platform, which has been evolving since 2017. This includes an automated method for measuring precision without relying on manually inspected true bug baselines, enhancing transparency, and identifying regressions during continuous integration. Additionally, the introduction of a hybrid inlining framework addresses the challenges of context sensitivity in inter-procedural static analysis by selectively inlining critical statements while summarizing non-critical ones, resulting in efficient analysis of large Java programs with minimal time overhead. Furthermore, a survey on Intermediate Representations (IR) underscores their role in enhancing static analysis by providing essential program information and enabling diverse analyses across different programming languages, while also revealing opportunities for further research in IR design. [24, 41, 14]

To enhance computational efficiency, it is crucial to investigate and implement a variety of methodologies, such as performance modeling, static analysis, and program decomposition, which can significantly improve performance across diverse applications by optimizing code through better understanding of program behavior and reducing the empirical search space in autotuning processes. [41, 24, 25, 110, 31]. One of the key aspects of this optimization involves the integration of domain-specific knowledge into neural networks, which serves to refine the processing capabilities and resource management strategies utilized in these systems. This integration not only facilitates a

AI-generated, for reference only.

more efficient handling of large datasets but also aligns computational processes with the unique requirements of specific domains.

9.1 Domain-Specific Knowledge in Neural Networks

9.2 Domain-Specific Knowledge in Neural Networks

The application of domain-specific knowledge is crucial in enhancing computational efficiency within neural networks, particularly when processing large datasets. Techniques such as the Dynamic Parallel Processing Framework (DPPF) leverage parallel processing to dynamically adjust resource allocation based on real-time data analysis, ensuring efficient handling of extensive data volumes [114]. This approach contrasts with traditional static partitioning methods by adapting the partitioning strategy according to data characteristics, thereby optimizing resource utilization and improving overall computational efficiency [105].

Incorporating adaptive learning techniques, as seen in the Adaptive Real-Time Data Processing (ARTDP) method, further enhances computational efficiency by optimizing processing parameters based on real-time data feedback. This dynamic optimization allows neural networks to maintain performance levels while minimizing resource consumption, making them more suitable for large-scale applications [64].

Furthermore, the integration of domain-specific knowledge into vulnerability detection frameworks, such as SATriage, has been shown to improve resource management by prioritizing vulnerabilities based on contextual risk. This advancement not only aids in efficient vulnerability detection but also contributes to the overall computational efficiency of security-oriented applications [115]. The emphasis on energy consumption estimations in static analysis also plays a significant role in achieving computational efficiency, particularly in embedded systems where resource constraints are critical [116].

Overall, the strategic application of domain-specific knowledge in neural networks and related fields underscores the importance of adaptive and context-aware approaches in optimizing computational efficiency. By utilizing real-time data insights and employing techniques that prioritize critical tasks, such as static analysis and optimized code generation, these methods enhance the operational efficiency of neural networks within the limitations of contemporary computing environments. This approach not only improves algorithm performance by leveraging specialized optimizations but also addresses challenges like data leakage and the high false-positive rates in static analysis warnings, ultimately ensuring that neural networks can effectively harness available computational resources. [117, 22, 118, 119, 62]

9.3 Resource Management in Large Dataset Processing

Efficient resource management in the processing of large datasets is a critical challenge in modern computing, necessitating strategies that optimize computational resources while maintaining performance. One approach involves leveraging parallel and distributed processing techniques to manage large-scale data efficiently. For instance, the Dynamic Parallel Processing Framework (DPPF) utilizes parallel processing to dynamically adjust resource allocation based on real-time data analysis, ensuring that computational resources are utilized effectively [114]. This method contrasts with traditional static partitioning by adapting the partitioning strategy according to data characteristics, thereby optimizing resource utilization [105].

Moreover, the Adaptive Real-Time Data Processing (ARTDP) method exemplifies the integration of adaptive learning techniques into resource management strategies. ARTDP optimizes processing parameters based on real-time data feedback, allowing for dynamic adjustments that maintain performance levels while minimizing resource consumption [64]. This approach is particularly beneficial in large-scale applications where data characteristics can vary significantly, requiring flexible resource management solutions.

In addition to these techniques, the incorporation of domain-specific knowledge into data processing frameworks can further enhance resource management. For example, the use of energy consumption estimations in static analysis helps optimize resource allocation in embedded systems, where resource constraints are critical [116]. By prioritizing tasks and optimizing resource usage, these frameworks ensure that large datasets are processed efficiently without compromising performance.

AI-generated, for reference only.

Overall, effective resource management in large dataset processing involves a combination of parallel processing, adaptive learning, and domain-specific optimizations. By implementing these advanced static analysis strategies, developers can optimize the utilization of computational resources, facilitating the efficient processing of large datasets while minimizing resource consumption and maximizing performance. For instance, leveraging frameworks that analyze file-processing programs in conjunction with input file format specifications can enhance the precision of program verification and transformation tasks. Additionally, static analysis frameworks tailored for data science notebooks can address challenges related to correctness and reproducibility, ensuring that the majority of notebooks execute analyses in under a second. Furthermore, tools like ROSA can significantly improve the performance and memory efficiency of R programs by identifying key program properties that enable various optimizations, leading to notable reductions in execution time and resource usage. [7, 22, 44]

9.4 Parallel Processing and Cloud Resource Management

Parallel processing and cloud resource management are integral components in enhancing computational efficiency, particularly in the context of large-scale data processing and complex computational tasks. These strategies leverage the concurrent execution of tasks across multiple processors or distributed systems, thereby optimizing resource utilization and improving scalability. In parallel processing, techniques such as the Dynamic Parallel Processing Framework (DPPF) dynamically adjust resource allocation based on real-time data analysis, ensuring efficient handling of extensive data volumes [114]. This approach contrasts with traditional static partitioning methods by adapting the partitioning strategy according to data characteristics, thereby optimizing resource utilization [105].

Cloud resource management further augments computational efficiency by providing scalable and flexible resource allocation tailored to the specific needs of applications. The integration of cloud computing resources, such as those provided by platforms like Microsoft Azure and Amazon Web Services (AWS), facilitates the dynamic scaling of computational power and storage capabilities. This adaptability allows organizations to effectively manage varying workloads by automatically allocating resources as needed, thereby enhancing performance and ensuring that computational and storage resources are readily available during peak demand periods. Additionally, leveraging cloud services can improve the efficiency of static and dynamic analysis processes, as demonstrated in various studies, including those focused on static analysis for microservices and Python applications utilizing cloud SDKs. [32, 2, 120, 121, 90]. This flexibility is crucial for applications with fluctuating demands, as it minimizes resource wastage and optimizes cost-effectiveness.

Moreover, the use of cloud-based platforms enables the deployment of parallel processing techniques at a larger scale, facilitating the execution of complex tasks across distributed environments. This integration enhances the efficient processing of large datasets by utilizing the computational power of multiple cloud instances, which significantly improves performance and reduces processing times. By implementing on-demand data-flow analysis techniques, particularly through the use of optimal and perfectly parallel algorithms, the system can swiftly handle specific program queries while leveraging lightweight preprocessing phases. This approach not only accelerates query responses but also facilitates embarrassingly parallel processing, allowing multiple threads to operate concurrently with minimal workload per thread. Additionally, the integration adheres to best practices for cloud services, ensuring robust type inference and static analysis that further optimize performance and reliability in data processing tasks. [121, 22, 21]. The synergy between parallel processing and cloud resource management is further exemplified by the use of adaptive algorithms that optimize resource allocation based on real-time feedback, ensuring that both computational and storage resources are utilized efficiently.

Overall, the strategic combination of parallel processing and cloud resource management plays a pivotal role in optimizing computational efficiency. By effectively combining automated static analysis techniques with advanced code completion strategies, developers can enhance the capability of software systems to efficiently manage large-scale data processing tasks. This integration not only minimizes resource consumption but also maximizes performance, allowing for improved quality assurance and defect detection in production code. For instance, the STALL+ framework demonstrates how static analysis can be seamlessly incorporated into repository-level code completion, yielding significant improvements in both effectiveness and efficiency across various programming languages.

AI-generated, for reference only.

Such collaborative approaches empower developers to leverage the strengths of each methodology, ensuring robust and high-performing software solutions. [79, 110]

9.5 Adaptive Algorithms for Data Processing

Adaptive algorithms play a critical role in enhancing data processing efficiency by dynamically adjusting computational strategies based on real-time data characteristics and system feedback. These algorithms are particularly valuable in environments where data volume and complexity can vary significantly, necessitating flexible and responsive processing solutions. The Adaptive Real-Time Data Processing (ARTDP) method exemplifies this approach by optimizing processing parameters based on real-time data feedback, allowing for continuous adjustments that maintain performance levels while minimizing resource consumption [64]. This dynamic optimization capability ensures that data processing systems can effectively handle fluctuations in data characteristics, leading to improved computational efficiency.

In addition to ARTDP, the integration of machine learning techniques into adaptive algorithms further enhances their effectiveness. By leveraging data-driven insights, these algorithms can refine execution paths and resource allocation, ensuring that processing strategies are aligned with current data conditions. This approach not only optimizes resource utilization but also enhances the overall performance of data processing systems, making them more suitable for large-scale applications [105].

Moreover, adaptive algorithms are instrumental in the context of parallel and distributed processing, where they facilitate the efficient management of computational resources across multiple processors or nodes. By dynamically adjusting task allocation and execution strategies, these algorithms ensure that parallel processing systems operate at optimal efficiency, reducing processing times and improving scalability [114]. This adaptability is crucial for handling large datasets and complex computational tasks, where traditional static processing methods may fall short.

Overall, the use of adaptive algorithms in data processing represents a significant advancement in computational efficiency. By continuously optimizing processing strategies through a dual-phase approach that leverages both static and dynamic analyses based on real-time data insights, these algorithms enable software systems to adeptly adapt to fluctuating data conditions. This results in significant improvements in performance and resource utilization within modern computing environments, as evidenced by frameworks like SODA, which enhances execution speed by up to 60

9.6 Challenges in Computational Efficiency for ROS Frameworks

Achieving computational efficiency within the Robot Operating System (ROS) frameworks presents several challenges, particularly in the context of parallel and distributed computing environments. One significant challenge is the accurate estimation of execution times across different platforms, which is crucial for ensuring that robotic applications can operate efficiently and reliably. The variability in execution environments and hardware configurations complicates the prediction of execution times, making it difficult to optimize resource allocation and scheduling in ROS-based systems [122].

Additionally, the handling of non-linear local ranking functions poses a challenge in enhancing the scalability and applicability of static analysis methods within ROS frameworks. These functions are critical for optimizing the performance of concurrent and recursive programs, which are prevalent in robotic applications. Future research aims to address these challenges by developing more sophisticated techniques for managing non-linear behaviors and improving the integration of static analysis methods in ROS environments [98].

The dynamic nature of Robot Operating System (ROS) applications, characterized by their reliance on real-time data processing and adaptive behaviors, significantly complicates the pursuit of computational efficiency, particularly in safety-critical robotic systems where software quality and reliability are essential. This complexity is exacerbated by the limited availability of tailored analysis mechanisms for assessing ROS2 code, as current research predominantly focuses on ROS1. Furthermore, the intricate and varied functionalities of ROS applications necessitate sophisticated static and dynamic analysis techniques to ensure robust performance and precision, highlighting the need for innovative approaches to improve software verification in these environments. [13, 123]. The need for responsive and flexible processing solutions requires the development of adaptive algorithms

AI-generated, for reference only.

that can adjust to changing data conditions and computational demands. These algorithms must be capable of optimizing resource utilization while maintaining the performance and reliability of ROS-based systems.

The challenges associated with achieving computational efficiency within Robot Operating System (ROS) frameworks underscore the critical need for ongoing innovation and research, particularly in the realms of software quality assessment, static analysis integration, and the development of tailored analysis mechanisms for both ROS1 and ROS2. This necessity is emphasized by findings that reveal significant gaps in the current research landscape, especially regarding the reliability and performance of ROS software in safety-critical applications, as well as the varying effectiveness of static analysis tools across different programming languages and contexts. [40, 41, 123, 23, 110]. By addressing the complexities of execution time estimation, non-linear function handling, and adaptive processing, developers can enhance the performance and scalability of ROS applications, ensuring that they meet the demands of modern robotic environments.

9.7 Enhancing Efficiency in Large-Scale Machine Learning

Enhancing computational efficiency in large-scale machine learning is a critical area of focus, aiming to optimize resource utilization and improve processing speeds. One promising approach involves the use of parallel processing frameworks, which distribute data across multiple processing units. This strategy significantly enhances efficiency and reduces processing time, making it an effective solution for handling the vast data volumes inherent in machine learning applications [114]. The Dynamic Parallel Processing Algorithm (DPPA) exemplifies this approach, demonstrating substantial improvements in processing speed and scalability over traditional methods, thereby confirming its robustness in addressing big data challenges within machine learning contexts [105].

Furthermore, the Adaptive Data Processing Algorithm (ADPA) has shown considerable promise in large-scale data processing tasks. Experimental results indicate that ADPA outperforms conventional methods in both efficiency and adaptability, providing a flexible solution that can dynamically adjust to varying data characteristics [63]. This adaptability is crucial for maintaining optimal performance in machine learning systems, where data properties can fluctuate significantly.

Future research endeavors aim to extend these frameworks to accommodate a broader range of programming paradigms and effects, including non-termination and probabilistic computations. Such advancements will further enhance the applicability and effectiveness of these frameworks in diverse machine learning environments, ensuring that they remain at the forefront of computational efficiency innovations [124]. By leveraging these advanced processing techniques, large-scale machine learning projects can achieve greater efficiency, scalability, and adaptability, meeting the growing demands of modern data-intensive applications.

10 Conclusion

10.1 Future Directions and Innovations

The trajectory of software analysis and optimization is poised for transformative advancements through targeted research and novel methodologies. A key focus should be on refining static analysis tools to enhance their precision and applicability, particularly in automated program repair systems. This involves advancing bug detection pattern mining and developing sophisticated patch generation techniques. Expanding the analytical scope and automating further transformations within frameworks like ROSA can significantly boost optimization capabilities, offering more comprehensive solutions for software performance enhancement.

Optimizing the implementation of languages like Datalog and extending analyses to encompass the complex features of languages such as Scheme will be critical. These efforts will improve the precision and scalability of static analysis methods, enabling their application across a variety of programming environments. In the realm of secure software development, enhancing the accuracy of Static Application Security Testing (SAST) warnings and employing ensemble approaches that integrate multiple SASTs will bolster secure code reviews, offering robust protection against vulnerabilities.

Innovative techniques for dynamic analysis are essential, particularly in the context of asynchronous JavaScript, to improve data race detection and strengthen the robustness of JavaScript analysis

AI-generated, for reference only.

tools. This includes extending static analysis to accommodate new asynchronous features, ensuring comprehensive evaluations of modern web applications. Enhancing tools for control and data flow execution in complex Java environments will broaden capabilities and improve usability for a wider audience.

The variability of bug injection points in systems such as BUG-INJECTOR should be enhanced to support concurrency-related bug injection, thereby improving the realism and effectiveness of benchmarking efforts. This will lead to more accurate assessments of static analysis tool performance. Moreover, integrating machine learning techniques to enhance malware detection capabilities and addressing current methodological limitations will improve security assessments and software reliability.

Collectively, these future directions and innovations hold the promise of addressing existing limitations and opening new research and development avenues. By leveraging advancements in static and dynamic analysis, machine learning, and adaptive algorithms, the field of software analysis and optimization will continue to evolve, offering more precise and effective solutions for contemporary software engineering challenges.

www.SurveyX.cn

AI-generated, for reference only.

References

[1] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects, 2020.

[2] Simon Schneider, Alexander Bakhtin, Xiaozhou Li, Jacopo Soldani, Antonio Brogi, Tomas Cerny, Riccardo Scandariato, and Davide Taibi. Comparison of static analysis architecture recovery tools for microservice applications, 2024.

[3] Ming-Ho Yee, Ayaz Badouraly, Ondřej Lhoták, Frank Tip, and Jan Vitek. Precise dataflow analysis of event-driven applications, 2019.

[4] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review, 2024.

[5] Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro. Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification, 2021.

[6] Safeeullah Soomro, Zainab Alansari, and Mohammad Riyaz Belgaum. Control and data flow execution of java programs, 2017.

[7] Rathijit Sen, Jianqiao Zhu, Jignesh M. Patel, and Somesh Jha. Rosa: R optimizations with static analysis, 2017.

[8] Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous javascript programs, 2019.

[9] Davis Ross Silverman, Yihao Sun, Kristopher Micinski, and Thomas Gilray. So you want to analyze scheme programs with datalog?, 2021.

[10] Jukka Ruohonen, Mubashrah Saddiqa, and Krzysztof Sierszecki. A static analysis of popular c packages in linux, 2024.

[11] Omar I. Al-Bataineh, Anastasiia Grishina, and Leon Moonen. Towards more reliable automated program repair by integrating static analysis techniques, 2021.

[12] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models, 2023.

[13] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts (extended version), 2021.

[14] Jiangchao Liu, Jierui Liu, Peng Di, Diyu Wu, Hengjie Zheng, Alex Liu, and Jingling Xue. Hybrid inlining: A compositional and context sensitive static analysis framework, 2022.

[15] Bingbing Rao, Zixia Liu, Hong Zhang, Siyang Lu, and Liqiang Wang. Soda: A semantics-aware optimization framework for data-intensive applications using hybrid program analysis, 2021.

[16] Gustaf Holst and Felix Dobslaw. On the importance and shortcomings of code readability metrics: A case study on reactive programming, 2021.

[17] Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. On-the-fly static analysis via dynamic bidirected dyck reachability, 2023.

[18] Damian M. Lyons, Anne Marie Bogar, and David Baird. Lightweight multilingual software analysis, 2018.

[19] Johannes Mey, Thomas Kühn, René Schöne, and Uwe Aßmann. Reusing static analysis across different domain-specific languages using reference attribute grammars, 2020.

AI-generated, for reference only.

www.SurveyX.cn

[20] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte. Automated customized bug-benchmark generation, 2019.

[21] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Optimal and perfectly parallel algorithms for on-demand data-flow analysis, 2020.

[22] Pavle Subotić, Lazar Milikić, and Milan Stojić. A static analysis framework for data science notebooks, 2021.

[23] Florian Sihler, Lukas Pietzschmann, Raphael Straub, Matthias Tichy, Andor Diera, and Abdelhalim Dahou. On the anatomy of real-world r code for static analysis, 2024.

[24] Bowen Zhang, Wei Chen, Hung-Chun Chiu, and Charles Zhang. Unveiling the power of intermediate representations for static analysis: A survey, 2024.

[25] Kewen Meng and Boyana Norris. Guiding optimizations with meliora: A deep walk down memory lane, 2020.

[26] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification, 2024.

[27] Vincenzo Arceri and Isabella Mastroeni. Static program analysis for string manipulation languages, 2019.

[28] Waqas Aman. A framework for analysis and comparison of dynamic malware analysis tools, 2014.

[29] Osejobe Ehichoya and Chinwuba Christian Nnaemeka. Evaluation of static analysis on web applications, 2022.

[30] Fabrizio Pastore and Leonardo Mariani. Dynamic analysis of regression problems in industrial systems: Challenges and solutions, 2017.

[31] Ali Reza Ibrahimzada. Program decomposition and translation with static analysis, 2024.

[32] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18Th international working conference on source code analysis and manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[33] Newsha Ardalani, Urmish Thakker, Aws Albarghouthi, and Karu Sankaralingam. A static analysis-based cross-architecture performance prediction using machine learning, 2019.

[34] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in java-based open source software development, 2023.

[35] Gabor Horvath, Reka Kovacs, Richard Szalay, Zoltan Porkolab, Gyorgy Orban, and Daniel Krupp. Static code analysis with codechecker, 2024.

[36] Amjed Tahir and Stephen G. MacDonell. Combining dynamic analysis and visualization to explore the distribution of unit test suites, 2021.

[37] Roberto Natella. Evaluation of systems programming exercises through tailored static analysis, 2024.

[38] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Fast static analyses of software product lines – an example with more than 42,000 metrics, 2021.

[39] Aaron Beigelbeck, Maurício Aniche, and Jürgen Cito. Interactive static software performance analysis in the ide, 2021.

[40] Baijun Cheng, Cen Zhang, Kailong Wang, Ling Shi, Yang Liu, Haoyu Wang, Yao Guo, Ding Li, and Xiangqun Chen. Semantic-enhanced indirect call analysis with large language models, 2024.

AI-generated, for reference only.

www.SurveyX.cn

[41] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Easing maintenance of academic static analyzers, 2024.

[42] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. Are automated static analysis tools worth it? an investigation into relative warning density and external software quality, 2021.

[43] Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. Using microservice telemetry data for system dynamic analysis, 2022.

[44] Raveendra Kumar Medicherla, Raghavan Komondoor, and S. Narendran. Static analysis of file-processing programs using file format specifications, 2015.

[45] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator, 2019.

[46] Rahma Mahmood and Qusay H. Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code, 2018.

[47] Marcos Dosea, Claudio Sant'Anna, Ythanna Oliveira, and Methanias Colaco Junior. A survey of software code review practices in brazil, 2020.

[48] Riasat Mahbub, Mohammad Masudur Rahman, and Muhammad Ahsanul Habib. On the prevalence, evolution, and impact of code smells in simulation modelling software, 2024.

[49] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd, 2023.

[50] Avi Hayoun, Veselin Raychev, and Jack Hair. Customizing static analysis using codesearch, 2024.

[51] William R. Nichols Jr au2. The cost and benefits of static analysis during development, 2020.

[52] Igor Regis da Silva Simões and Elaine Venson. Evaluating source code quality with large language models: a comparative study, 2024.

[53] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues, 2023.

[54] Ehsan Mashhadi, Shaiful Chowdhury, Somayeh Modaberi, Hadi Hemmati, and Gias Uddin. An empirical study on bug severity estimation using source code metrics and static analysis, 2024.

[55] Pranav Garg and Srinivasan Sengamedu SHS. Example-based synthesis of static analysis rules, 2022.

[56] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[57] Peter Sun and Dae-Kyoo Kim. Analyzing impact of dependency injection on software maintainability, 2022.

[58] Daniel Kulesz and Jan-Peter Ostberg. Practical challenges with spreadsheet auditing tools, 2014.

[59] Lori Flynn, William Snavely, and Zachary Kurtz. Test suites as a source of training data for static analysis alert classifiers, 2021.

[60] Mario Gleirscher, Dmitriy Golubitskiy, Maximilian Irlbeck, and Stefan Wagner. Introduction of static quality analysis in small and medium-sized software enterprises: Experiences from technology transfer, 2016.

AI-generated, for reference only.

www.SurveyX.cn

[61] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes, 2022.

[62] Lawrence McAfee and Kunle Olukotun. Utilizing static analysis and code generation to accelerate neural networks, 2012.

[63] Étienne Payet, Fred Mesnard, and Fausto Spoto. Non-termination analysis of java bytecode, 2014.

[64] Jean-Loup Carre and Charles Hymans. From single-thread to multithreaded: An efficient static analysis algorithm, 2009.

[65] Jones Yeboah and Saheed Popoola. Efficacy of static analysis tools for software defect detection on open-source projects, 2024.

[66] Lisa Nguyen Quang Do and Eric Bodden. Explaining static analysis with rule graphs. *IEEE Transactions on Software Engineering*, 48(2):678–690, 2020.

[67] Diogo Silveira Mendonça and Marcos Kalinowski. An empirical investigation on the challenges of creating custom static analysis rules for defect localization, 2021.

[68] Colin S. Gordon. Synthesizing program-specific static analyses, 2018.

[69] Eric Bodden. Self-adaptive static analysis, 2017.

[70] Mohammad Amin Alipour, Alex Groce, Chaoqiang Zhang, Anahita Sanadaji, and Gokul Caushik. Finding model-checkable needles in large source code haystacks: Modular bug-finding via static analysis and dynamic invariant discovery, 2016.

[71] Daniel de Carvalho, Manuel Mazzara, Bogdan Mingela, Larisa Safina, Alexander Tchitchigin, and Nikolay Troshkov. Jolie static type checker: a prototype, 2017.

[72] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel c programs, 2012.

[73] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. How to find actionable static analysis warnings: A case study with findbugs, 2022.

[74] Tim Sonnekalb, Christopher-Tobias Knaust, Bernd Gruner, Clemens-Alexander Brust, Lynn von Kurnatowski, Andreas Schreiber, Thomas S. Heinze, and Patrick Mäder. A static analysis platform for investigating security trends in repositories, 2023.

[75] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis-a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32. IEEE, 2019.

[76] Yoshiki Higo, Shinpei Hayashi, Hideaki Hata, and Meiyappan Nagappan. Ammonia: An approach for deriving project-specific bug patterns, 2020.

[77] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. Static detection of uninitialized stack variables in binary code, 2020.

[78] Ivan R. Ivanov, Joachim Meyer, Aiden Grossman, William S. Moses, and Johannes Doerfert. Input-gen: Guided generation of stateful inputs for testing, tuning, and training, 2024.

[79] Mario Gleirscher, Dmitriy Golubitskiy, Maximilian Irlbeck, and Stefan Wagner. On the benefit of automated static analysis for small and medium-sized software enterprises, 2016.

[80] Isabella Mastroeni and Vincenzo Arceri. Improving dynamic code analysis by code abstraction, 2021.

[81] Flash Sheridan. Static analysis deployment pitfalls, 2022.

[82] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of the monolith, 2020.

AI-generated, for reference only.

www.SurveyX.cn

[83] Tiago Matias, Filipe F. Correia, Jonas Fritzsch, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining microservice boundaries: A case study using static and dynamic software analysis, 2020.

[84] Muhammad Numair Mansur, Benjamin Mariano, Maria Christakis, Jorge A. Navas, and Valentin Wüstholz. Automatically tailoring static analysis to custom usage scenarios, 2020.

[85] Atanu Barai, Nandakishore Santhi, Abdur Razzak, Stephan Eidenbenz, and Abdel-Hameed A. Badawy. Llvm static analysis for program characterization and memory reuse profile estimation, 2023.

[86] Hongki Lee, Changhee Park, and Sukyoung Ryu. Automatically tracing imprecision causes in javascript static analysis, 2019.

[87] Ivan Postolski, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Dynamic slicing by on-demand re-execution, 2022.

[88] Safeeullah Soomro, Zahid Hussain, and Ayaz Keerio. Path conditions help to locate and localize faults from programs, 2013.

[89] Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static analysis of infrastructure as code: a survey, 2022.

[90] Tomas Cerny and Davide Taibi. Static analysis tools in the era of cloud-native systems, 2022.

[91] Abdur Razzak, Atanu Barai, Nandakishore Santhi, and Abdel-Hameed A. Badawy. Static reuse profile estimation for array applications, 2024.

[92] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis, 2015.

[93] Patrícia Monteiro, João Lourenço, and António Ravara. Uma análise comparativa de ferramentas de análise estática para deteção de erros de memória, 2018.

[94] Chengpeng Wang, Yifei Gao, Wuqi Zhang, Xuwei Liu, Qingkai Shi, and Xiangyu Zhang. Llmsa: A compositional neuro-symbolic approach to compilation-free and customizable static analysis, 2024.

[95] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the extensional scalability problem for value-flow analysis frameworks, 2019.

[96] Anna-Katharina Wickert, Michael Schlichtig, Marvin Vogel, Lukas Winter, Mira Mezini, and Eric Bodden. Supporting error chains in static analysis for precise evaluation results and enhanced usability, 2024.

[97] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543, 2022.

[98] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis, 2014.

[99] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. Abstract interpretation of binary code with memory accesses using polyhedra, 2017.

[100] Luke Panayi, Rohan Gandhi, Jim Whittaker, Vassilios Chouliaras, Martin Berger, and Paul Kelly. Improving memory dependence prediction with static analysis, 2024.

[101] Simon Wegener, Kris K. Nikov, Jose Nunez-Yanez, and Kerstin Eder. Energyanalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications, 2023.

[102] Gabriel Kerneis, Charlie Shepherd, and Stefan Hajnoczi. Qemu/cpc: Static analysis and cps conversion for safe, portable, and efficient coroutines, 2013.

AI-generated, for reference only.

www.SurveyX.cn

[103] Ruba Mutasim, Gabriel Synnaeve, David Pichardie, and Baptiste Rozière. Leveraging static analysis for bug repair, 2023.

[104] David Monniaux and Cyril Six. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion, 2021.

[105] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for java, 2010.

[106] Bart van Oort, Luís Cruz, Babak Loni, and Arie van Deursen. "project smells" – experiences in analysing the software quality of ml projects with mllint, 2022.

[107] Riyadh Baghdadi, Albert Cohen, Cedric Bastoul, Louis-Noel Pouchet, and Lawrence Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization, 2011.

[108] Linnea Stjerna and David Broman. Programming with context-sensitive holes using dependency-aware tuning, 2022.

[109] Valentina Lenarduzzi, Savanna Lujan, Nyyti Saarimaki, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision, 2021.

[110] Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. Stall+: Boosting llm-based repository-level code completion with static analysis, 2024.

[111] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[112] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. Call graph soundness in android static analysis, 2024.

[113] Eduardo Rosales, Matteo Basso, Andrea Rosà, and Walter Binder. Profiling and optimizing java streams, 2023.

[114] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. Failure-directed program trimming (extended version), 2017.

[115] Yan Wu, Jingyi Su, David D. Moran, and Chris D. Near. Automated software testing starting from static analysis: Current state of the art, 2023.

[116] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for llvm ir programs, 2015.

[117] Filip Drobnjaković, Pavle Subotić, and Caterina Urban. Abstract interpretation-based data leakage static analysis, 2024.

[118] Thanh Trong Vu and Hieu Dinh Vo. Using multiple code representations to prioritize static analysis warnings, 2022.

[119] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a static analyzer from data, 2017.

[120] Rahul Kumar, Chetan Bansal, and Jakob Lichtenberg. Static analysis using the cloud, 2016.

[121] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. Static analysis for aws best practices in python code, 2022.

[122] Edison Mera, Pedro Lopez-Garcia, German Puebla, Manuel Carro, and Manuel Hermenegildo. Towards execution time estimation for logic programs via static analysis and profiling, 2007.

[123] Mohannad Alhanahnah. Software quality assessment for robot operating system, 2020.

[124] Sandra Alves, Delia Kesner, and Miguel Ramos. Quantitative global memory, 2023.

AI-generated, for reference only.

www.sunnyx.cn

Disclaimer:

SurveyX is an AI-powered system designed to automate the generation of surveys. While it aims to produce high-quality, coherent, and comprehensive surveys with accurate citations, the final output is derived from the AI's synthesis of pre-processed materials, which may contain limitations or inaccuracies. As such, the generated content should not be used for academic publication or formal submissions and must be independently reviewed and verified. The developers of SurveyX do not assume responsibility for any errors or consequences arising from the use of the generated surveys.

www.SurveyX.cn

AI-generated, for reference only.