An-Nahdhah
JOURNAL OF EDUCATION AND CULTURE

Vol. 2 No. 1 (2026)

Available online at
https://ejournal.arate.id/index.php/An-Nahdhah

Leveraging Functional Programming for Real-Time Systems: A Literature Review

Eddie Jr S. Cenita[1]

[1] North Eastern Mindanao State University - Lianga Campus, Philippines

Article History

Received: 2026/07/08
Revised: 2026/11/15
Accepted: 2026/11/23
Published: 2026/11/24

Keywords

Functional Programming; Real-Time Systems; Worst-Case Execution Time; Memory Management; Immutability; Temporal Predictability.

Corresponding Author:

Eddie Jr S. Cenita

eddiecenitajr143@gmail.com

DOI: *https://doi.org/10.63199/an-nahdhah.v2i1.96*

Real-time systems impose stringent temporal requirements in which the correctness of computation depends not only on the accuracy of results but also on their timely execution. Although functional programming (FP) offers important advantages through immutability, referential transparency, and compositional design, its suitability for real-time computing remains constrained by challenges associated with execution-time predictability, memory management, and resource utilization. This study systematically reviews the application of functional programming in real-time systems, focusing on the relationship between functional abstractions and temporal predictability. The retrieved from IEEE Xplore, ACM Digital Library, Scopus, SpringerLink, and Google Scholar. Following predefined eligibility criteria, the selected studies were analyzed using thematic synthesis, with particular attention to programming languages and runtime environments, application domains, memory management strategies, concurrency mechanisms, and real-time performance characteristics. The synthesis identifies three interconnected dimensions: the benefits of functional abstractions for controlling shared state and improving concurrency reasoning; the implementation-related challenges associated with memory allocation, garbage collection, and runtime overhead; and the limitations of establishing temporal guarantees, particularly worst-case execution time (WCET) and deadline compliance. The findings suggest that immutability and referential transparency can improve reasoning about program behavior but do not independently guarantee deterministic execution or hard real-time schedulability. The practical suitability of functional programming therefore depends on runtime architecture, memory management strategies, compiler support, and the temporal constraints of the target application. This review contributes an analytical framework for evaluating functional programming in real-time environments and identifies research priorities concerning WCET-aware compilation, bounded memory allocation, predictable garbage collection, and hybrid programming architectures.

Copyright © 2026 by the authors. This is an open access publication under the terms and conditions of the Creative Commons Attribution (CC BY NC 4.0) license.

1. Introduction

Real-time systems are computing systems in which correctness depends not only on the accuracy of computational results but also on whether those results are produced within specified time constraints. Unlike conventional computing systems, where performance is frequently evaluated in terms of throughput and average execution time, real-time systems must satisfy temporal requirements that are determined by the operational characteristics of their applications. A missed deadline may result in different levels of consequences depending on the system's criticality. In hard real-time systems, missing a deadline constitutes a system failure and may lead to severe safety consequences, whereas firm real-time systems can tolerate occasional late results only when those results no longer have operational value. In soft real-time systems, deadline violations generally result in degraded service quality rather than complete system failure. These distinctions are particularly important in application domains such as automotive control, avionics, medical devices, industrial automation, and embedded computing, where the temporal behavior of software directly affects system reliability and operational safety (Burns & Wellings, 2009; Minaeva & Hanzálek, 2021).

The development of reliable real-time systems requires careful consideration of programming languages, compiler behavior, runtime environments, memory management, and task scheduling mechanisms. Imperative programming languages, particularly C and assembly, have traditionally been employed in resource-constrained and safety-critical applications because they provide relatively direct control over memory, hardware resources, and execution behavior. However, their suitability does not arise solely from low-level hardware access. Predictable execution also depends on compiler optimizations, operating system support, available development tools, worst-case execution time (WCET) analysis, and compliance with applicable safety requirements. As software systems become increasingly complex and concurrent, alternative programming paradigms have attracted attention for their potential to improve program structure, maintainability, and reasoning about system behavior. One such paradigm is Functional Programming (FP), which emphasizes pure functions, referential transparency, immutable data, and compositional program design (Murphy et al., 2019).

Functional Programming provides a computational model in which program behavior is expressed primarily through function evaluation rather than the modification of shared program state. Its emphasis on immutability and referential transparency can simplify reasoning about program behavior, reduce unintended side effects, and facilitate the development of modular software components. These

properties are particularly relevant to concurrent and distributed applications, where shared mutable state can introduce data races and complicate the verification of program behavior. Functional abstractions may also support fault isolation and compositional reasoning by making dependencies between computational components more explicit. Nevertheless, these advantages should not be interpreted as automatic guarantees of temporal correctness. Immutability can reduce certain concurrency-related risks, but it does not independently eliminate deadlocks, starvation, synchronization problems, or unpredictable execution delays. Similarly, referential transparency improves reasoning about computational results but does not establish bounded execution time or guarantee that a task will complete before its deadline.

The central challenge in applying Functional Programming to real-time systems is therefore the relationship between its high-level computational abstractions and the strict temporal and resource constraints imposed by real-time applications. Although functional languages offer expressive mechanisms for describing computations, their runtime implementations may introduce additional considerations, including dynamic memory allocation, garbage collection, persistent data structure allocation, closure creation, and runtime scheduling overhead. These mechanisms can influence execution-time variability, memory consumption, and response-time predictability, particularly in resource-constrained embedded environments. Consequently, evaluating the suitability of FP requires more than comparing average execution performance or examining its theoretical advantages. It also requires investigating worst-case execution time (WCET), worst-case response time, scheduling predictability, interrupt latency, execution jitter, memory bounds, and schedulability analysis. Previous research has examined the real-time capabilities of functional languages and compilers, including their runtime characteristics and dynamic memory management mechanisms. Murphy et al. (2019), for example, surveyed functional language compilers and evaluated their applicability to real-time embedded systems. Their work provides an important foundation for understanding the relationship between functional language design and real-time requirements. However, further analytical synthesis is needed to connect language-level properties, runtime implementation strategies, memory management approaches, and temporal guarantees across different real-time application contexts.

In addition, the applicability of Functional Programming should be examined across different implementation environments and categories of real-time constraints. Languages such as Haskell, Erlang, and OCaml provide different approaches to functional abstractions, concurrency, and runtime execution. Their practical suitability

cannot be generalized without considering the specific language implementation, compiler and runtime configuration, target hardware, memory allocation strategy, and application requirements. Functional Reactive Programming (FRP) also represents a related but distinct research direction that applies functional abstractions to time-varying values and event-driven computations. Its relevance to embedded and reactive systems warrants consideration, although FRP should not be treated as synonymous with Functional Programming in general. Furthermore, high-performance computing and low-latency distributed applications should be distinguished from hard real-time systems because high throughput or low average latency does not necessarily establish bounded worst-case execution or deadline compliance. These distinctions are essential for evaluating empirical evidence and avoiding overly broad conclusions about the effectiveness of functional approaches in time-critical computing environments.

Against this background, this study conducts a systematic literature review to examine the opportunities, limitations, and implementation considerations of Functional Programming in real-time systems. Rather than treating immutability or functional abstraction as sufficient evidence of real-time suitability, the review focuses on the interaction between programming language characteristics, runtime mechanisms, resource management, concurrency, and temporal performance requirements. Specifically, the study addresses three research questions:

RQ1**:** How do the fundamental characteristics of Functional Programming, particularly immutability, referential transparency, and compositionality, contribute to program reasoning, concurrency management, and reliability in real-time systems? RQ2**:** What technical challenges and implementation mechanisms associated with functional languages and their runtime environments affect temporal predictability, memory management, and the fulfillment of real-time constraints? RQ3**:** Under what application conditions, system configurations, and implementation strategies can Functional Programming be effectively adopted in real-time systems, and what approaches can address its limitations?

By addressing these questions, this review aims to develop an analytical understanding of the relationship between functional programming principles and the technical requirements of real-time computing. Its contribution lies in synthesizing evidence across programming language characteristics, runtime implementation, memory management, concurrency mechanisms, and temporal performance considerations. The review also identifies research priorities concerning WCET-aware compilation, bounded memory allocation, predictable garbage collection, resource-aware functional abstractions, and hybrid programming architectures that combine

high-level functional reasoning with low-level control over time-critical operations. Through this synthesis, the study seeks to clarify not only the potential advantages of Functional Programming but also the technical conditions under which its application to real-time systems is appropriate.

2. Literature Review

2.1. Real-Time Systems and Temporal Constraints

Real-time systems are computing systems in which computational correctness is determined by both the validity of results and the time at which those results are produced. Unlike conventional computing environments, where execution efficiency is commonly evaluated through average processing time and throughput, real-time systems require predictable responses within predefined temporal constraints. The consequences of failing to meet these constraints depend on the operational requirements of the system. Hard real-time systems require strict compliance with deadlines, whereas firm real-time systems tolerate late results only when their operational usefulness has expired. Soft real-time systems, in contrast, permit occasional deadline violations that generally result in reduced service quality rather than catastrophic system failure (Burns & Wellings, 2009).

Several technical characteristics determine the reliability of real-time systems. These include responsiveness, bounded resource consumption, concurrency, and distributed communication. Responsiveness concerns the ability of a system to respond to external events within specified time limits. Bounded resource consumption requires predictable utilization of processing time, memory, and other computational resources. Concurrency involves the coordinated execution of multiple tasks, while distributed communication introduces additional timing considerations associated with network delays, synchronization, and information exchange. These characteristics are interconnected because variations in resource utilization and task execution can influence the overall response time of a system.

A fundamental requirement in hard real-time computing is the ability to establish an upper bound on task execution time. Worst-case execution time (WCET) analysis estimates the maximum time required for a particular task to execute under specified hardware and software conditions. This information is subsequently used in schedulability analysis to determine whether the system can complete its tasks within their respective deadlines. Consequently, average execution time and high throughput are insufficient indicators of hard real-time suitability. Other relevant parameters include worst-case response time, execution-time variability, jitter, interrupt latency, and priority inversion. These parameters must be evaluated in relation to the

scheduling policy, processor architecture, operating system, and characteristics of the application (Minaeva & Hanzálek, 2021).

The distinction between computational efficiency and temporal predictability is particularly important when evaluating alternative programming paradigms. A programming language may provide expressive abstractions and efficient average execution while still introducing unpredictable runtime behavior. Dynamic memory allocation, garbage collection, complex compiler optimizations, and asynchronous task execution can influence the temporal characteristics of an application. Therefore, the suitability of a programming language for real-time systems should be assessed not only through its computational expressiveness but also through its ability to support bounded execution, predictable resource utilization, and reliable scheduling.

2.2. Fundamental Principles of Functional Programming

Functional Programming (FP) is a programming paradigm that expresses computation primarily through the evaluation and composition of functions. Its theoretical foundations are closely associated with lambda calculus, which provides a mathematical framework for representing function abstraction and application. Unlike imperative programming, which commonly describes computation through sequences of state-changing instructions, FP emphasizes function evaluation, immutable data, and explicit relationships between inputs and outputs. Hughes (1989) explains that functional programming supports modular program construction by enabling complex operations to be developed through the composition of smaller functions.

One of the fundamental characteristics of FP is referential transparency, which means that an expression can be replaced by its corresponding value without changing the observable behavior of the program, provided that the expression is evaluated under the same conditions. This property facilitates reasoning about program behavior and supports equational reasoning. Pure functions, which produce consistent outputs for identical inputs without modifying external state, also reduce dependencies between computational components. Such characteristics are particularly useful in complex software systems because they make program behavior easier to analyze, test, and maintain.

Another important characteristic is immutability, which refers to the inability to modify an existing data object after its creation. Instead of changing existing values, operations produce new values or structures representing the updated state. Immutability can simplify program reasoning and reduce the possibility of unintended modifications to shared data. However, immutability should not be treated as a universal property of every functional language or implementation. The

extent of immutability may depend on language design, type-system restrictions, runtime behavior, and programming conventions. Haller and Axelsson (2017), for example, examine immutability in Scala and demonstrate the importance of distinguishing language-level guarantees from programming practices that encourage immutable data.

Functional programming also emphasizes higher-order functions, recursion, pattern matching, and declarative abstractions. Higher-order functions allow functions to be passed as arguments or returned as results, enabling flexible program composition. Pattern matching provides a structured mechanism for handling different data forms, while recursion supports the expression of repetitive computations without relying exclusively on mutable iteration variables. These mechanisms can improve program modularity and expressiveness, although their implementation may introduce additional runtime costs depending on compiler optimization and execution environments.

Despite these advantages, the theoretical properties of functional programming do not automatically guarantee real-time performance. Referential transparency does not establish an upper bound on execution time, while immutability does not necessarily eliminate allocation costs. Similarly, the termination of a program does not imply that it will terminate within a required deadline. Consequently, functional programming principles must be evaluated in relation to implementation-level properties, including execution-time bounds, memory consumption, compiler behavior, and runtime management.

2.3. Functional Programming and Concurrent Systems

Concurrency is an essential characteristic of modern real-time systems because multiple computational tasks may need to execute simultaneously or respond to external events independently. In imperative programming, concurrent tasks frequently interact through shared mutable data, which can introduce synchronization problems, race conditions, and inconsistent program states. Managing these interactions requires mechanisms such as locks, semaphores, and other synchronization primitives. Although these mechanisms can ensure controlled access to shared resources, they may also introduce additional execution overhead and complications in determining task response times.

Functional programming provides an alternative approach by emphasizing immutable data and reducing the need for shared mutable state. When multiple tasks access immutable values, they can generally read the same data without requiring synchronization to protect against modifications to those values. This characteristic can simplify concurrent program design and reduce certain categories of data races.

Furthermore, pure functions allow computational components to operate with fewer dependencies on external program states, facilitating modular reasoning and parallel execution.

However, the relationship between immutability and concurrency safety requires careful interpretation. Immutability can reduce data races involving shared mutable objects, but it does not eliminate all concurrency-related problems. Deadlocks may still occur when tasks compete for other shared resources, while starvation and priority inversion may arise from inappropriate scheduling and synchronization mechanisms. Moreover, immutable data structures may require additional memory allocation when new versions are created. Therefore, the effectiveness of functional programming in concurrent real-time environments depends on the interaction between data representation, task coordination, memory management, and scheduling mechanisms.

Functional programming languages provide different mechanisms for implementing concurrent computations. Erlang, for example, emphasizes isolated processes and message-passing communication, while Haskell provides abstractions such as software transactional memory (STM) for coordinating concurrent operations. These approaches offer different strategies for controlling interactions between computational components. Nevertheless, their real-time suitability must be evaluated according to measurable temporal properties rather than inferred solely from their concurrency models. In particular, synchronization delays, task scheduling, message-processing latency, and resource contention remain important considerations when assessing their use in deadline-constrained applications.

2.4. Memory Management and Runtime Predictability

Memory management represents one of the central challenges in applying functional programming to resource-constrained and real-time systems. Functional languages frequently employ dynamically allocated data structures to support immutable values, higher-order functions, closures, and recursive computations. Persistent data structures, for example, preserve previous versions of data while allowing new versions to be created through structural sharing. Although this approach provides advantages for program reasoning and state management, it may introduce allocation activity and additional memory consumption depending on the data structure and implementation.

Garbage collection is another important consideration because many functional language implementations rely on automatic memory reclamation. Garbage collection reduces the burden of manual memory management and can help prevent memory leaks and dangling references. However, its execution may introduce additional

runtime activity that influences application latency. In systems with strict temporal requirements, unpredictable collection pauses or variations in allocation behavior can complicate the establishment of reliable execution-time bounds.

The impact of memory management should not be attributed to immutability alone. Different sources of runtime overhead must be distinguished, including persistent data structure allocation, garbage collection, boxing, closure allocation, and runtime scheduling. Their effects depend on the language implementation, compiler optimization, allocation strategy, workload characteristics, and target hardware. Consequently, an immutable data representation does not necessarily impose a fixed performance penalty, particularly when compilers employ optimization techniques such as allocation elimination, unboxing, and efficient structural sharing.

Several approaches can be considered to improve memory predictability in real-time environments. These include static memory allocation, region-based memory management, bounded allocation mechanisms, and garbage collectors specifically designed to satisfy temporal constraints. Real-time garbage collection research has demonstrated that memory reclamation can be organized to reduce interference with time-critical tasks, although the effectiveness of such approaches depends on the underlying runtime architecture and scheduling assumptions. Murphy et al. (2019) provide an important contribution by examining functional language compilers and empirically investigating how dynamic memory management affects performance and predictability in real-time embedded environments.

The relationship between memory management and temporal predictability therefore requires an integrated assessment. A suitable implementation must control not only total memory consumption but also allocation frequency, reclamation behavior, fragmentation, and the maximum execution time associated with memory operations. These considerations are particularly important for embedded systems with limited memory capacity and strict response-time requirements.

2.5. **Functional Reactive Programming in Embedded Systems**

Functional Reactive Programming (FRP) is a programming paradigm designed to describe reactive systems through abstractions such as time-varying values, signals, and events. Although FRP is related to functional programming, it represents a more specialized approach for modeling systems that continuously respond to changes in their environments. Its declarative nature enables developers to describe relationships between changing values without explicitly managing every sequence of updates through conventional polling or callback mechanisms.

FRP has received attention in embedded system development because many embedded applications interact continuously with sensors, actuators, and external

events. Yokoyama et al. (2021) introduced Emfrp, a functional reactive programming language designed for small-scale embedded systems. The language employs restrictions on recursive data types and functions to support static determination of runtime memory requirements and guarantee termination of reactive updates. Their work subsequently introduced an extended type system that supports more expressive data structures while retaining these static properties. The study also presents a compiler targeting C and evaluates runtime overhead.

Further development of Emfrp demonstrates efforts to address the execution requirements of embedded reactive applications. Yokoyama et al. (2022) investigated the introduction of asynchronous tasks into an FRP language for small-scale embedded systems. Their approach addresses the need to express asynchronous computations while preserving important properties of the language's execution model. These developments indicate that FRP research is not limited to declarative descriptions of reactive behavior but also concerns the management of execution and resource constraints.

Suzuki et al. (2023) subsequently presented Emfrp-REPL, an interpreter for functional reactive programming on small embedded devices. Their evaluation on resource-constrained microcontrollers, including the ESP32 platform, reported memory usage comparable to MicroPython and a narrower latency range in the reported microbenchmarks. These findings provide practical evidence that functional reactive abstractions can be implemented on small embedded devices. However, the reported latency comparison should not be interpreted as proof of hard real-time schedulability because bounded worst-case execution and deadline guarantees require additional analysis.

These studies illustrate the importance of distinguishing FRP from general-purpose functional programming. FRP offers specialized abstractions for reactive and time-varying computations, while general FP encompasses a broader range of computational applications. Their contributions are relevant to real-time computing, but their suitability must be evaluated according to the guarantees provided by the language, compiler, and runtime implementation.

2.6. Functional Programming Languages and Real-Time Capabilities

The practical application of functional programming in real-time systems depends substantially on the characteristics of individual languages and their implementation environments. Languages such as Haskell, Erlang, and OCaml provide different computational models, runtime architectures, and approaches to memory management. Therefore, their real-time capabilities should not be generalized solely on the basis of their classification as functional programming languages.

Haskell provides strong static typing, lazy evaluation, higher-order functions, and extensive support for functional abstractions. These characteristics facilitate modular program design and sophisticated reasoning about computational behavior. However, lazy evaluation and runtime memory management may introduce additional considerations when establishing predictable execution times. The suitability of Haskell for real-time applications consequently depends on the compiler, runtime configuration, allocation behavior, and techniques used to control computational resources.

Erlang emphasizes lightweight processes, message passing, and fault-tolerant application architectures. Its concurrency model is particularly relevant to systems involving multiple independent processes and distributed communication. Nevertheless, the ability to support large numbers of concurrent processes does not by itself establish compliance with hard real-time deadlines. The runtime scheduler, message-processing behavior, garbage collection, and resource contention must also be considered when evaluating temporal performance.

OCaml provides a combination of functional abstractions, static typing, and imperative capabilities. Its compilation and runtime mechanisms support efficient execution in various application domains. Nevertheless, the presence of garbage collection and dynamic allocation requires attention when OCaml is considered for resource-constrained or deadline-critical applications. The evaluation should distinguish between general-purpose performance and the ability to establish reliable upper bounds on execution and memory consumption.

Research on functional languages and real-time embedded systems indicates that specialized language restrictions, compiler design, and runtime mechanisms can influence the extent to which functional abstractions satisfy temporal requirements. Murphy et al. (2019) provide a systematic examination of functional language compilers and their real-time capabilities, emphasizing the relationship between language characteristics, dynamic memory management, and predictable execution. More recent work on embedded FRP languages, including Emfrp, demonstrates how static memory analysis and restrictions on computational constructs can support resource-bounded execution (Yokoyama et al., 2021).

A related development is the use of synchronous functional or data-flow languages in safety-critical real-time applications. Scade, for example, represents a specialized approach to developing safety-critical software through synchronous and data-flow abstractions. Research on its semantics illustrates the importance of precisely defining the execution behavior of reactive computations and the relationship between high-level language constructs and executable implementations.

Such approaches demonstrate that real-time suitability can be supported through language design and formal execution models rather than through functional programming principles alone.

Overall, the literature indicates that functional programming offers valuable mechanisms for modularity, reasoning, and concurrency management. However, its applicability to real-time systems depends on the extent to which language abstractions can be translated into predictable runtime behavior. The interaction between immutability, memory management, execution-time analysis, and scheduling remains central to determining whether a particular functional implementation can satisfy the temporal requirements of its target application. This relationship provides the conceptual foundation for the present review and guides the subsequent analysis of functional programming benefits, implementation challenges, and practical adoption strategies.

3. Methods

3.1.Research Design

This study employed a qualitative systematic literature review (SLR) to investigate the application, capabilities, and limitations of Functional Programming (FP) in real-time systems. The review was designed to identify, evaluate, and synthesize existing research concerning the relationship between functional programming principles and the technical requirements of real-time computing. Particular attention was given to immutability, referential transparency, concurrency management, memory allocation, runtime behavior, and temporal predictability.

The review process was structured according to established systematic review practices in software engineering, particularly the guidelines proposed by Kitchenham and Charters (2007). PRISMA 2020 was additionally adopted as a reporting framework to improve transparency in the identification, screening, eligibility assessment, and inclusion of relevant studies (Page et al., 2021). These guidelines were used to establish a reproducible review procedure and ensure that the selection of literature was based on predefined eligibility criteria rather than subjective relevance alone.

The review addressed three research questions concerning: (1) the contribution of functional programming principles to program reasoning, concurrency management, and reliability; (2) the technical challenges associated with runtime behavior, memory management, and temporal predictability; and (3) the implementation conditions and strategies under which functional programming can be adopted in real-time environments.

A qualitative synthesis approach was selected because the identified studies were expected to differ in programming languages, runtime architectures, experimental environments, performance indicators, and application domains. Consequently, a statistical meta-analysis was not considered appropriate for the heterogeneous evidence examined in this review.

3.2.Information Sources and Search Strategy

The literature search was designed to identify relevant studies addressing the intersection of functional programming and real-time computing. Five academic databases were selected: IEEE Xplore, ACM Digital Library, Scopus, SpringerLink, and Google Scholar. These databases were chosen to cover research in computer science, programming language theory, software engineering, embedded systems, and real-time computing.

The search strategy incorporated three major groups of concepts: functional programming, real-time computing, and implementation-related characteristics. Synonyms and related technical terms were combined using Boolean operators to improve the identification of relevant publications.

The principal search string was formulated as follows: Search String: ("functional programming" OR "functional language" OR Haskell OR Erlang OR OCaml OR "functional reactive programming" OR FRP) AND ("real-time system" OR "real-time systems" OR "hard real-time" OR "soft real-time" OR "real-time programming" OR embedded) AND (immutability OR "memory management" OR concurrency OR "worst-case execution time" OR WCET OR scheduling OR latency OR predictability)

The search string was adapted to the syntax and search capabilities of each database. Searches were conducted using title, abstract, and keyword fields where supported. For platforms with different indexing and search interfaces, equivalent combinations of terms were applied.

The review considered publications from the inception of the relevant databases until [insert actual final search date]. No arbitrary starting year was imposed in order to retain foundational contributions to real-time functional programming while also capturing contemporary developments in compiler technology, embedded systems, and runtime optimization. The search was restricted to English-language publications to maintain consistency in the evaluation and synthesis of the selected literature.

Google Scholar was additionally used to identify relevant publications not retrieved through the primary database searches and to examine the reference lists of selected studies. To maintain transparency, the search date, search terms, and screening limits applied to Google Scholar were documented separately.

3.3.Inclusion and Exclusion Criteria

To ensure the relevance and consistency of the selected literature, predefined inclusion and exclusion criteria were established before the screening process. Studies were considered eligible for inclusion when they met the following criteria: Publications explicitly examining functional programming, functional languages, or functional reactive programming in relation to real-time systems, embedded computing, or time-sensitive applications.

Studies investigating functional programming characteristics such as immutability, referential transparency, concurrency, memory management, or runtime behavior with implications for real-time computing. Research examining execution-time predictability, worst-case execution time, scheduling, latency, memory bounds, or other temporal performance characteristics of functional language implementations. Peer-reviewed journal articles and conference proceedings presenting theoretical contributions, empirical evaluations, experimental implementations, or relevant case studies. Publications available in English with sufficient methodological or technical information to support data extraction and critical analysis.

Studies were excluded when they met one or more of the following conditions: Publications unrelated to functional programming or real-time computing. Studies discussing general-purpose functional programming without a meaningful connection to real-time, embedded, or time-sensitive applications. Publications addressing conventional performance optimization without examining relevant temporal or resource constraints. Duplicate publications or multiple records referring to the same study, unless a later publication provided substantially different findings.

Editorials, opinion articles, commercial promotional materials, and other non-peer-reviewed sources that did not provide sufficient academic or technical evidence. Publications without accessible full text or sufficient information to evaluate their relevance to the research questions. Foundational theoretical publications were also considered when necessary to establish the conceptual basis of functional programming and real-time computing. However, these sources were distinguished from empirical primary studies during the synthesis process.

3.4.Study Selection Process

The study selection process followed the identification, screening, eligibility, and inclusion stages recommended by PRISMA 2020. Initially, records were retrieved from the selected databases using the predefined search strategy. All retrieved records were consolidated into a reference management system to facilitate organization and duplicate identification.

Duplicate records were removed before the screening stage. The remaining records were screened based on their titles and abstracts to determine their potential relevance to the research questions. Publications that appeared relevant or contained insufficient information for an initial decision were retained for full-text assessment.

During the eligibility assessment, the full texts of potentially relevant publications were examined against the predefined inclusion and exclusion criteria. Studies that did not provide sufficient evidence concerning functional programming and real-time computing were excluded, with the primary reasons for exclusion documented.

To improve the reliability of study selection, the screening process was conducted by [insert actual number of reviewers] reviewers. Disagreements were resolved through discussion and, where necessary, consultation with an additional reviewer. The final set of eligible publications constituted the evidence base for the qualitative synthesis.

The identification and selection process was documented using a PRISMA 2020 flow diagram, reporting the number of records identified, duplicates removed, records screened, full-text publications assessed, exclusions, and studies included in the final review.

3.5.Data Extraction and Quality Assessment

A structured data extraction framework was developed to ensure consistency in the collection and comparison of information from the selected publications. The extraction process focused on characteristics directly related to the research questions.

The following information was extracted from each included study: Bibliographic information: author names, publication year, publication type, and research venue. Programming language and implementation: functional language, compiler, runtime environment, and relevant implementation characteristics. Application domain: embedded systems, industrial automation, distributed applications, reactive systems, and other real-time computing environments. Real-time characteristics: hard, firm, or soft deadline requirements, responsiveness, and temporal constraints. Memory management: garbage collection, static allocation, dynamic allocation, region-based memory management, and persistent data structures. Concurrency mechanisms: immutable data, message passing, synchronization, parallel execution, and task scheduling. Performance indicators: execution time, WCET, response time, latency, jitter, memory consumption, and other reported resource measurements. Research findings and limitations: principal contributions, experimental results, identified constraints, and recommendations for future research.

In addition to data extraction, the methodological and technical quality of the included studies was assessed using criteria adapted to the nature of software engineering research. The assessment considered the clarity of research objectives, adequacy of the research design, transparency of implementation details, appropriateness of evaluation methods, relevance of reported performance indicators, and acknowledgment of study limitations.

Each criterion was evaluated using a three-level scale: 1 (Yes), 0.5 (Partially), and 0 (No). The resulting assessment was used to characterize the strength and limitations of the available evidence rather than to assume that every included study provided equivalent empirical support.

Studies were retained or excluded according to their relevance and evidential contribution, with quality assessment results considered during the interpretation of the synthesis.

3.6.Data Analysis and Synthesis

The extracted data were analyzed using thematic synthesis and qualitative content analysis. This approach was selected to identify recurring concepts, technical relationships, implementation strategies, and differences in research findings across the selected publications.

The analysis was conducted through three interconnected stages. First, the extracted information was organized according to the research questions and the predefined data extraction categories. This stage involved identifying recurring concepts associated with functional programming principles, concurrency, memory management, runtime implementation, and temporal performance.

Second, relevant findings were coded and grouped into broader analytical themes. The initial coding framework was informed by the research questions, while additional categories were developed when new technical issues emerged from the included studies. Particular attention was given to distinguishing theoretical properties of functional programming from implementation-dependent performance characteristics.

Third, findings across the included studies were compared to identify common patterns, contrasting results, and limitations in the available evidence. The synthesis examined how different programming languages, compiler implementations, runtime architectures, and memory management strategies influenced the suitability of functional programming for real-time applications.

The final synthesis was organized into three principal analytical dimensions corresponding to the research questions: Functional programming principles and system reliability: examining immutability, referential transparency, modularity,

concurrency management, and their contributions to program reasoning and reliability. Implementation challenges and temporal predictability: examining memory allocation, garbage collection, runtime overhead, WCET, scheduling, and resource constraints. Practical applicability and implementation strategies: examining programming language characteristics, embedded implementations, hybrid architectures, and conditions under which functional programming can satisfy real-time requirements.

Through this process, the review sought to distinguish between conceptual advantages, empirically demonstrated implementation benefits, and unresolved technical limitations. The resulting synthesis provides the basis for evaluating the applicability of functional programming in different categories of real-time systems and identifying directions for future research.

4. Results and Discussion

4.1. Overview of the Reviewed Literature

The reviewed literature reveals that research on Functional Programming (FP) in real-time systems has developed along several interconnected directions, including real-time language design, compiler and runtime optimization, memory management, and the implementation of functional abstractions in embedded environments. Rather than demonstrating a universal advantage of functional programming, the available studies indicate that its applicability depends substantially on the mechanisms used to control execution behavior and computational resources.

Murphy et al. (2019) provide an important foundation through their examination of functional language compilers and their applicability to real-time embedded systems. Their research investigates the characteristics required of real-time programming languages and evaluates the influence of dynamic memory management on runtime predictability and performance. This work establishes that the suitability of functional programming must be assessed at both the language and implementation levels.

Subsequent research has investigated how general-purpose functional languages can be adapted for real-time applications. Shivkumar et al. (2021) presented Real-time MLton, an adaptation of the Standard ML compiler and runtime environment for embedded and real-time applications. Their work involved modifications to the threading subsystem, object model, and garbage collector, together with a preliminary implementation of a radar-based aircraft collision detection application. This demonstrates that real-time functional programming can

require substantial modifications to the runtime architecture rather than relying exclusively on conventional functional language implementations.

Another research direction focuses on domain-specific functional reactive languages for resource-constrained embedded systems. Yokoyama et al. (2021) developed an extended type system for Emfrp, enabling recursive data types while maintaining static memory determination and termination guarantees for reactive updates. Their research illustrates how restrictions in language design can support resource predictability in embedded environments.

The development of Emfrp was subsequently extended to asynchronous computation. Yokoyama et al. (2022) investigated mechanisms for introducing asynchronous tasks into the language to address the requirements of embedded applications involving operations with different execution characteristics. Suzuki et al. (2023) further examined Emfrp-REPL, an interpreter designed for small embedded devices. Their evaluation demonstrated that the interpreter could operate on resource-constrained microcontrollers, with memory consumption comparable to MicroPython and a narrower latency range in the reported microbenchmarks.

In addition to language and runtime development, research on functional programming has explored the possibility of establishing formal execution-cost guarantees. Work on the Hume language, for example, investigated the combination of source-level cost models and low-level WCET analysis to support verifiable resource-usage guarantees in real-time embedded applications (Hammond et al., 2006).

Taken together, these studies indicate that research on functional programming in real-time environments can be classified into three principal directions: (1) adapting general-purpose functional language runtimes to real-time requirements; (2) developing restricted or domain-specific functional languages for embedded applications; and (3) establishing formal methods and analytical techniques for bounding execution costs and resource consumption. These directions provide the empirical and conceptual basis for the subsequent synthesis of functional programming principles, implementation challenges, and practical applicability in real-time systems.

4.2. **Functional Programming and the Reliability of Real-Time Systems**

The findings indicate that functional programming (FP) offers several characteristics that may support the development of reliable real-time systems, particularly immutability, referential transparency, and compositionality. These characteristics can improve the way developers reason about program behavior, manage state changes, and construct concurrent computations. However, their

contribution to real-time reliability should be understood as an enabling factor rather than a direct guarantee of temporal correctness.

Immutability reduces the possibility of unintended state modifications because data objects are not modified after their creation. In concurrent applications, this characteristic can reduce certain forms of shared-state interference and simplify the reasoning process when multiple computations operate on the same data. Nevertheless, immutability does not eliminate all concurrency problems. Synchronization, resource contention, scheduling interference, and communication delays may still occur, particularly when functional programs interact with hardware, operating-system services, or mutable external resources.

The Real-time MLton study illustrates how functional programming can be adapted to practical real-time requirements. Rather than assuming that a conventional functional runtime is automatically suitable for real-time execution, Shivkumar et al. (2021) modified important runtime components, including the threading subsystem, object model, and garbage collector. This implementation demonstrates that functional programming can be brought closer to real-time application requirements through deliberate runtime engineering. It also indicates that the suitability of a functional language depends not only on its programming constructs but also on the behavior of its compiler and runtime environment.

This finding is consistent with the broader survey by Murphy et al. (2019), which identifies real-time capabilities in functional languages and compilers as an important area requiring attention to execution behavior and runtime implementation. Therefore, the reliability benefits of FP should be evaluated together with the mechanisms responsible for executing programs under temporal constraints.

4.3. Temporal Predictability and Execution-Time Analysis

A central issue emerging from the reviewed literature is the distinction between program correctness and temporal predictability. A program may produce logically correct outputs but still fail as a real-time system if those outputs are delivered after their deadlines. Consequently, the evaluation of FP in real-time environments must extend beyond functional correctness to include worst-case execution time (WCET), response time, execution-time variation, and schedulability.

The work on Hume by Hammond et al. (2006) is particularly relevant because it explores the relationship between functional programming and formally verifiable WCET analysis. Its approach combines source-level cost models with information obtained from lower-level program analysis. This demonstrates that execution-time analysis for functional programs requires an understanding of both high-level language constructs and the behavior of the generated machine code.

The findings suggest that temporal predictability cannot be inferred simply from the use of pure functions or immutable data structures. Compiler transformations, runtime services, memory allocation, garbage collection, and hardware interactions may influence execution time. A functional program that relies on dynamic allocation or unpredictable runtime operations may still exhibit significant execution-time variation.

The implications are especially important for hard real-time systems, in which missing a deadline can cause unacceptable consequences. In such environments, developers need defensible upper bounds on execution time and resource consumption. For firm and soft real-time applications, occasional deadline misses may have different levels of acceptability, but response-time variation and deadline compliance remain important evaluation criteria.

Accordingly, the adoption of FP in real-time systems should be accompanied by suitable timing-analysis techniques. Static analysis, bounded execution models, predictable runtime configurations, and measurements under representative workloads can contribute to this objective. However, empirical measurements alone should not be interpreted as formal proof of WCET because observed execution times do not necessarily establish an upper bound for all possible execution conditions.

4.4. Memory Management as a Critical Implementation Challenge

Memory management represents one of the most significant implementation challenges identified in the reviewed studies. Functional programming languages frequently employ dynamic data structures, automatic memory management, and garbage collection. These mechanisms can improve programming convenience and reduce certain memory-management errors, but they may introduce execution-time variability.

The Real-time MLton study provides an important example of addressing this issue through runtime modification. Its implementation demonstrates that the garbage collector and object-management mechanisms must be considered when adapting a functional language to real-time applications (Shivkumar et al., 2021). This is important because an unpredictable garbage-collection pause can interfere with the timely execution of critical tasks.

A different approach is demonstrated by Emfrp, a functional reactive programming language designed for small-scale embedded systems. Yokoyama et al. (2021) investigated the use of recursive data types while addressing memory-related constraints and termination properties in reactive computations. This direction is significant because it considers resource constraints as part of the language design rather than treating them solely as runtime problems.

The subsequent development of Emfrp-REPL further illustrates the practical importance of lightweight execution environments. Suzuki et al. (2023) evaluated an interpreter for functional reactive programming on small embedded devices, including an ESP32-based platform. Their evaluation reported memory consumption comparable to MicroPython in the examined configuration and a narrower latency range in the reported microbenchmarks. These findings are promising, although they should not be generalized to all embedded workloads or interpreted as proof of hard real-time guarantees.

Taken together, these studies indicate two broad implementation strategies. The first is to adapt an existing functional language runtime to provide more predictable memory behavior. The second is to develop a domain-specific language or execution environment designed around the limited resources of embedded devices. The choice between these strategies depends on application complexity, hardware capacity, timing requirements, and the level of predictability required.

4.5. Concurrency and Reactive Programming in Embedded Environments

Concurrency is another important dimension in the relationship between FP and real-time systems. Functional programming can simplify some aspects of concurrent computation by reducing shared mutable state. However, real-time concurrency also requires predictable task execution, communication, synchronization, and scheduling.

The reviewed work on asynchronous tasks in Emfrp represents an attempt to extend functional reactive programming toward more flexible embedded computation (Yokoyama et al., 2022). This development is relevant because embedded applications increasingly need to respond to multiple events and process data from different sources. Nevertheless, introducing asynchronous execution also raises questions about task coordination, execution ordering, and resource contention.

The Synchron approach provides another example of functional techniques being explored for embedded systems. Its implementation and evaluation consider an embedded execution environment and performance characteristics such as response time and jitter. Such measurements are relevant because they move the discussion beyond language-level properties toward observable behavior on target hardware.

These findings suggest that FP can contribute to the organization and reasoning of concurrent programs, but the benefits depend on the execution model. A system must still define how tasks are scheduled, how events are handled, how shared hardware resources are accessed, and how timing constraints are maintained. Thus, functional programming should be viewed as one component of a broader real-time

software architecture rather than a replacement for real-time scheduling and synchronization mechanisms.

4.6. Implications for the Adoption of Functional Programming

The reviewed studies suggest that the suitability of FP for real-time systems varies according to the application domain and the required level of temporal assurance. For hard real-time applications, the most important requirements are bounded execution time, predictable memory behavior, and verifiable deadline compliance. Functional programming may be adopted in such environments when the language implementation, compiler, runtime, and hardware platform support the necessary analysis and resource constraints. The Hume research and Real-time MLton demonstrate relevant approaches, but they do not establish that every functional language is suitable for hard real-time applications.

For firm and soft real-time applications, the adoption of FP may be more flexible, depending on the consequences of deadline misses. Embedded monitoring, reactive control interfaces, sensor processing, and other event-driven applications may benefit from functional and reactive abstractions. Nevertheless, response-time variation and resource consumption should still be evaluated in the intended operating environment.

The findings also imply that language selection should not be based solely on programming style or theoretical properties. Developers should examine the availability of compiler support, runtime configuration, memory-management mechanisms, timing-analysis tools, and hardware integration. A language with strong functional abstractions but limited control over runtime behavior may be less appropriate for applications with strict timing constraints than a more specialized implementation.

Therefore, the adoption of FP is most promising when its software-engineering benefits are combined with explicit real-time design practices. These include bounded resource allocation, predictable task execution, appropriate scheduling strategies, and systematic timing evaluation.

4.7. Research Gaps and Future Directions

Although the reviewed studies demonstrate several approaches to integrating FP with real-time and embedded systems, important research gaps remain. First, further research is needed on the formal analysis of execution time in functional programs. Existing work on Hume illustrates the potential of combining source-level cost models with low-level analysis. However, broader evaluation across contemporary functional languages, compilers, and hardware platforms is necessary to understand the general applicability of these techniques.

Second, memory-management strategies require further investigation. Runtime modification, garbage-collection control, static memory allocation, and lightweight domain-specific execution environments represent different approaches, but their comparative performance under equivalent real-time workloads remains an important research question.

Third, concurrency mechanisms need more systematic evaluation. Future studies should examine how functional abstractions interact with real-time scheduling, asynchronous tasks, interrupt handling, synchronization, and resource contention. Evaluation should include response-time bounds, jitter, and schedulability rather than relying exclusively on functional correctness or average execution time.

Fourth, more comprehensive experimental comparisons are needed. Existing studies examine different languages, platforms, workloads, and evaluation methods. This diversity makes direct comparison difficult. Future research should establish common benchmarks and evaluation criteria that include WCET, memory footprint, response time, jitter, and deadline-miss behavior.

Finally, research should investigate practical adoption in application domains with different levels of temporal criticality. Such investigations could clarify which functional programming techniques are appropriate for hard, firm, and soft real-time applications and what additional engineering mechanisms are required in each context.

4.8. Answers to the Research Questions

The findings provide the following answers to the three research questions.

RQ1: How do functional programming characteristics contribute to reasoning, concurrency, and reliability in real-time systems? Immutability, referential transparency, and compositionality can improve program reasoning, reduce unintended state changes, and simplify certain forms of concurrent computation. These properties may support software reliability, but they do not independently guarantee deadline compliance or temporal correctness. Their effectiveness depends on the implementation and execution environment.

RQ2: What technical challenges and implementation mechanisms affect temporal predictability, memory management, and the fulfillment of real-time constraints? The main challenges concern execution-time analysis, dynamic memory allocation, garbage collection, runtime behavior, and concurrent task coordination. Relevant implementation mechanisms include runtime modification, static or bounded memory strategies, domain-specific embedded languages, and formal or measurement-based timing analysis. The evidence indicates that these mechanisms must be evaluated together with compiler behavior and hardware characteristics.

RQ3: Under what application conditions and implementation strategies can functional programming be effectively adopted in real-time systems? FP can be considered for real-time applications when its language and runtime characteristics are compatible with the application's timing and resource requirements. Runtime adaptation may be appropriate for more general-purpose functional languages, while lightweight domain-specific languages may be suitable for resource-constrained embedded devices. The level of acceptable timing variation and the consequences of missing deadlines should guide language selection and implementation decisions.

Overall, the discussion indicates that functional programming is neither inherently unsuitable nor automatically suitable for real-time systems. Its practical applicability depends on the combination of language characteristics, compiler implementation, runtime behavior, memory-management strategy, timing-analysis methods, and the temporal requirements of the target application.

5. Conclusion

This study examined the applicability of functional programming (FP) in real-time systems by focusing on its programming characteristics, implementation challenges, and strategies for meeting temporal and resource constraints. The findings indicate that FP offers several potential benefits for real-time software development, particularly through immutability, referential transparency, and compositionality. These characteristics can improve program reasoning, reduce unintended state modifications, and simplify certain aspects of concurrent software development. However, they do not independently guarantee temporal predictability or compliance with real-time deadlines.

The analysis also identifies execution-time predictability and memory management as major challenges in adopting FP for real-time applications. Dynamic memory allocation, garbage collection, compiler behavior, and runtime mechanisms can introduce execution-time variation. Research on Real-time MLton, Hume, and embedded functional reactive programming demonstrates that these challenges can be addressed through different approaches, including runtime modification, execution-cost analysis, and the development of lightweight domain-specific languages. Nevertheless, the effectiveness of these approaches depends on the target hardware, application workload, and required level of temporal assurance.

The findings further suggest that FP adoption should be determined by the temporal criticality and resource requirements of the intended application. For hard real-time systems, predictable execution, bounded memory consumption, and reliable WCET analysis are essential. For firm and soft real-time systems, functional and

reactive abstractions may offer greater flexibility, provided that response-time behavior and resource consumption remain appropriate for the application. Therefore, the suitability of FP should be assessed through the combined characteristics of the language, compiler, runtime environment, memory-management strategy, and real-time execution architecture.

Despite these promising developments, the reviewed literature reveals several areas requiring further investigation. Future research should prioritize formal WCET analysis for functional languages, predictable memory-management mechanisms, the integration of functional concurrency with real-time scheduling, and standardized benchmarking across different languages and embedded platforms. More systematic empirical comparisons are also needed to establish the conditions under which FP can satisfy different categories of real-time requirements.

This study has limitations. The conclusions are based on the scope and characteristics of the literature examined and should not be interpreted as evidence that all functional programming languages or implementations provide equivalent real-time capabilities. Differences in hardware platforms, programming languages, runtime architectures, workloads, and evaluation methods also limit direct comparisons among the reviewed studies. Further systematic research involving broader database coverage, transparent study-selection procedures, and standardized evaluation criteria is necessary to strengthen the generalizability of the findings.

In conclusion, functional programming should not be regarded as inherently incompatible with real-time systems. Rather, its applicability depends on how its programming advantages are combined with predictable runtime behavior, controlled resource management, and rigorous temporal analysis. With appropriate implementation strategies and verification mechanisms, FP can provide a viable foundation for selected real-time and embedded applications while offering opportunities for further research into reliable and predictable software systems.

References

Burns, A., & Wellings, A. (2009). *Real-time systems and programming languages: Ada, real-time Java and C/real-time POSIX* (4th ed.). Addison-Wesley.

Grogono, P. D. (1988). Real-time functional programming. In R. Radharamanan (Ed.), *Robotics and factories of the future '87: Proceedings of the Second International Conference, San Diego, California, USA, July 28–31, 1987* (pp. 743–750). Springer. https://doi.org/10.1007/978-3-642-73890-6_89

Haller, P., & Axelsson, L. (2017). Quantifying and explaining immutability in Scala. *Electronic Proceedings in Theoretical Computer Science, 246*, 21–27. https://doi.org/10.4204/EPTCS.246.5

Hammond, K. (2005). Is it time for real-time functional programming? In S. Gilmore (Ed.), *Trends in functional programming* (Vol. 4, pp. 1–18). Intellect Books.

Hammond, K., Ferdinand, C., Heckmann, R., Dyckhoff, R., Hofmann, M., Jost, S., Loidl, H.-W., Michaelson, G., Pointon, R., Scaife, N., Sérot, J., & Wallace, A. (2006). Towards formally verifiable WCET analysis for a functional programming language. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET 2006)* (Vol. 4). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.WCET.2006.677

Hu, Z., Hughes, J., & Wang, M. (2015). How functional programming mattered. *National Science Review, 2*(3), 349–370. https://doi.org/10.1093/nsr/nwv042

Kitchenham, B., & Charters, S. (2007). *Guidelines for performing systematic literature reviews in software engineering* (Version 2.3; EBSE Technical Report EBSE-2007-01).

Minaeva, A., & Hanzálek, Z. (2021). Survey on periodic scheduling for time-triggered hard real-time systems. *ACM Computing Surveys, 54*(1), Article 23. https://doi.org/10.1145/3431232

Murphy, J. C., Shivkumar, B., Pritchard, A., Iraci, G., Kumar, D., Kim, S. H., & Ziarek, L. (2019). A survey of real-time capabilities in functional languages and compilers. *Concurrency and Computation: Practice and Experience, 31*(4), Article e4902. https://doi.org/10.1002/cpe.4902

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J. M., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilson, E., McDonald, S., ... Moher, D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ, 372*, n71. https://doi.org/10.1136/bmj.n71

Sarkar, A., Svensson, B. J., & Sheeran, M. (2022). Synchron—An API and runtime for embedded systems. In K. Ali & J. Vitek (Eds.), *36th European Conference on Object-Oriented Programming (ECOOP 2022)* (Vol. 222, pp. 17:1–17:29). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2022.17

Shivkumar, B., Murphy, J., & Ziarek, L. (2021). Real-time MLton: A Standard ML runtime for real-time functional programs. *Journal of Functional Programming, 31*, Article e19. https://doi.org/10.1017/S0956796821000174

Suzuki, G., Watanabe, T., & Moriguchi, S. (2023). Implementation and evaluation of an interpreter for functional reactive programming on small embedded devices. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (pp. 12–16). Association for Computing Machinery. https://doi.org/10.1145/3594671.3594674

Taha, W., Hudak, P., & Wan, Z. (2001). Directions in functional programming for real(-time) applications. In T. A. Henzinger & C. M. Kirsch (Eds.), *Embedded software* (pp. 185–203). Springer. https://doi.org/10.1007/3-540-45449-7_14

Yokoyama, A., Moriguchi, S., & Watanabe, T. (2021). A functional reactive programming language for small-scale embedded systems with recursive data types. *Journal of Information Processing, 29*, 685–706. https://doi.org/10.2197/ipsjjip.29.685

Yokoyama, A., Moriguchi, S., & Watanabe, T. (2022). Towards introducing asynchronous tasks to an FRP language for small-scale embedded systems. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems* (pp. 1–12). Association for Computing Machinery. https://doi.org/10.1145/3563837.3568338