Abstract

Embedded Systems application development has traditionally been carried out in low-level machine-oriented programming languages like C or Assembler that can result in unsafe, error-prone and difficult-to-maintain code. Functional programming with features such as higher-order functions, algebraic data types, polymorphism, strong static typing and automatic memory management appears to be an ideal candidate to address the issues with low-level languages plaguing embedded systems.

However, embedded systems usually run on heavily memory-constrained devices with memory in the order of hundreds of kilobytes and applications running on such devices embody the general characteristics of being (i) I/O-bound, (ii) concurrent and (iii) timing-aware. Popular functional language compilers and runtimes either do not fare well with such scarce memory resources or do not provide high-level abstractions that address all the three listed characteristics.

This work attempts to address this gap by investigating and proposing high-level abstractions specialised for I/O-bound, concurrent and timing-aware embedded-systems programs. We implement the proposed abstractions on eagerly-evaluated, statically-typed functional languages running natively on microcontrollers. Our contributions are divided into two parts -

Part 1 presents a functional reactive programming language - Hailstorm - that tracks side effects like I/O in its type system using a feature called resource types. Hailstorm's programming model is illustrated on the GRiSP microcontroller board.

Part 2 comprises two papers that describe the design and implementation of Synchron, a runtime API that provides a uniform message-passing framework for the handling of software messages as well as hardware interrupts. Additionally, the Synchron API supports a novel timing operator to capture the notion of time, common in embedded applications. The Synchron API is implemented as a virtual machine - SynchronVM - that is run on the NRF52 and STM32 microcontroller boards. We present programming examples that illustrate the concurrency, I/O and timing capabilities of the VM and provide various benchmarks on the response time, memory and power usage of SynchronVM.

Keywords

functional programming, embedded systems, virtual machine, concurrency, timing, language runtime

List of Publications

Appended publications

This thesis is based on the following publications:

[A] Abhiroop Sarkar, Mary Sheeran "Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications"
Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming. ACM, 2020.

[B] Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, Mary Sheeran "Higher-order concurrency for microcontrollers"
Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. ACM, 2021.

[C] Abhiroop Sarkar, Bo Joel Svensson, Mary Sheeran "Synchron - An API for Embedded Systems"
Under Review.

Research Contribution

Paper A

I was responsible for the research idea and implementation of the compiler presented in the paper. I wrote all the major sections of the paper with suggestions for the paper structure, edits and final enhancements made by my supervisor Mary Sheeran.

Paper B

I and Bo Joel Svensson, together, develop and maintain the virtual machine presented in the paper. I proposed the core research idea presented in the paper. I also designed and implemented the middleware, assembler, bytecode interpreter and major parts of the runtime. Bo Joel Svensson wrote the low-level bridge, the garbage collector and collaborated with me on several important design decisions within the runtime. The frontend language was written by Robert Krook.

I wrote all the major sections of the paper with final edits and enhancements provided by Bo Joel Svensson and Mary Sheeran. The experiments were conducted by Bo Joel Svensson and myself.

Paper C

This work builds on the work of Paper B and the contributions listed above apply here. I proposed the timing API and implemented the core parts of it within the runtime. The low-level timing subsystem was written by Bo Joel Svensson. The experiments presented in the paper were done by Bo Joel Svensson and myself.

I wrote the paper in collaboration with Bo Joel Svensson. Several edits and enhancements were proposed by Mary Sheeran.

Contents

Chapter 1

Introduction

Embedded Systems are ubiquitous artifacts of the digital age. From industrial machinery and smart buildings to automated highways and cars, embedded systems remains a driving force behind the automation of the world around us.

Unlike the traditional disciplines of batch computing and data processing, an embedded system is typically *embedded* within a larger system that involves interactions with the physical environment. In the light of this characteristic, Henzinger and Sifakis [1] defines an "embedded system" as given below -

Definition 1

An embedded system is an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (1) reaction to a physical environment, and (2) execution on a physical platform.

The first category of interactions gives rise to behavioural requirements on an embedded system application such as deadline, throughput, response time, etc., that can have a tangible impact on the physical environment. The physical interaction component demands that an embedded application be *reactive* to any stimulus provided by its environment.

On the other hand, the second category results in more implementation-specific requirements such as limited power usage, memory usage, etc. These constraints dictate the economics of embedded systems, which are deployed in large numbers in most applications areas (like sensor networks and cars) and require application development platforms that prioritise resource sensitivity over high performance.

The above discussion highlights two desired behaviours of embedded systems applications - (i) reactivity and (ii) resource sensitiveness. To delve into the design and implementation of languages and tools that can embody these behaviours, we need to understand the behaviours at a more operational level.

Reactivity

The word "reactive" is heavily overloaded, and it has been used to describe diverse programming models, libraries and frameworks. When we classify embedded systems as reactive in nature, we refer to the original definition of reactive systems, as presented by Harel and Pnueli [2] -

Definition 2

Reactive systems are those that are repeatedly prompted by the outside world and their role is to continuously respond to external inputs. A reactive system, in general, does not compute or perform a function, but is supposed to maintain a certain ongoing relationship, so to speak, with its environment.

If we compare the description of embedded systems from Definition 1 with the above definition, we can find parallels between the two. Additionally, the authors state that reactive systems do not lend themselves naturally to description in terms of functions and transformations.

Operationally, reactive applications are *I/O-intensive*, owing to their continual interactions with the external environment. On top of that, the external environment can supply a variety of external stimuli, which is best handled by breaking down an application into several *concurrent* stimulus handlers.

A third property that arises as a result of interaction with the external world is the notion of being *timing-aware*. Reactions to certain specific types of stimuli often requires responses within a given deadline and at a periodic rate. Hence, we can compile three important operational properties of reactive systems, which in turn is embodied in embedded systems application, as follows:

Fundamental Properties

1. I/O-intensive
2. Concurrent
3. Timing-aware

As an example of an embedded system that exhibits the above characteristics, let us consider a washing machine. It serves information to its user through an LED-based display while taking input from the user in the form of control knobs and buttons. The main function of the system is, however, to perform a wash cycle consisting of heating of water, filling the washing compartment with water, mixing in laundry detergent at the right time and dosage, spinning the drum at various speeds at various times and so on. All of this is accomplished through actuation via microcontroller peripherals such as a timer generating a Pulse-width Modulation (PWM) signal of the correct frequency and duty cycle to drive a motor at the desired speed or controlling relays for turning pumps on and off. All the while, sensors provide information to the microcontroller about clogged up filters or other non-ideal conditions. Overall, the application *concurrently* receives several *I/O impulses* while performing *time-bound* and *periodic* operations.

Resource Sensitiveness

Resource sensitivity drives the economics of embedded systems deployments. Consider a typical embedded systems application area like wireless sensor networks (WSNs), where the number of deployed devices ranges from hundreds to thousands. Such large deployments are made cost-effective by reducing the price of an individual unit to be in the range of 10 to 100 dollars.

The cost of these devices is cut down by manufacturing them to be heavily resource-constrained. Such devices, often microcontrollers, have a small die area with simple circuitry, missing components like on-chip cache, transistors for superscalar execution, etc. As a result, these devices are power efficient and require little cooling. They frequently use ARM-based microcontrollers, also with constrained memory and clock speed. So, we can summarise by saying -

> Embedded systems become cost-effective by using somewhat old, resource-constrained but high volume hardware.

Hence, any application development platform for embedded systems, whether it is a programming language or a runtime, needs to be designed in a resource-sensitive fashion. In practice, the platform should aim to operate with low power and memory usage, and support applications that can fulfil their tasks while running on a relatively weak processor.

At the same time, the growing cost of software development and security is a part of the resource sensitivity of embedded systems. Ravi et al. [3] propose that security is an additional dimension to consider in embedded systems, besides cost, power usage etc. Especially with internet connectivity among embedded devices, called Internet of Things (IoT), many more security challenges [4, 5] crop up.

In summary, programming embedded systems is a challenging task that involves designing *I/O-bound*, *concurrent* and *timing-aware* applications. Additionally, the applications should be resource-sensitive in terms of power and memory usage while accounting for the growing security challenges and software development costs. To understand the current state of programming such embedded applications, we shall next present a short survey on programming languages and frameworks used in embedded systems.

1.1 Embedded Systems Language Survey

The rapid proliferation of embedded systems has resulted in a large body of work, in both industry and academia, attempting to design embedded systems languages. Accordingly, we shall divide our survey into two parts.

1.1.1 Industrial Trends

The landscape of embedded systems language adoption was surveyed by VDC Research, in 2011 [6], by surveying engineers about the languages that they most frequently use at work. Fig. 1.1 shows the results of the survey.

Fig 1.1 shows that the C programming language had a major market share of embedded systems in 2011, followed by C++ and Assembly. Almost ten years

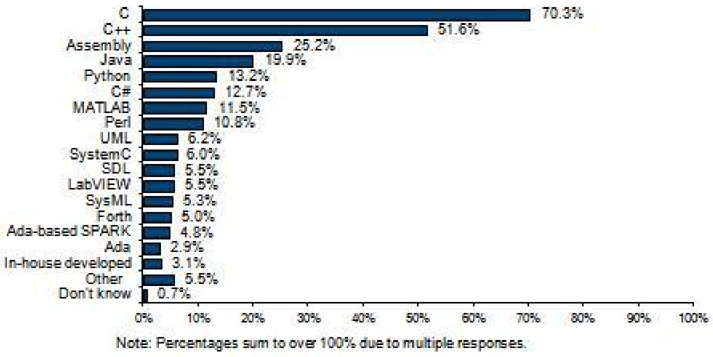

Figure 1.1: VDC 2011 Embedded Engineer Survey Results [6]

since then, a slightly different perspective (with Python overtaking Assembly language and Java) can be seen in the Embedded Markets Study conducted by EETimes in 2019 [7], shown in Fig. 1.2[1].

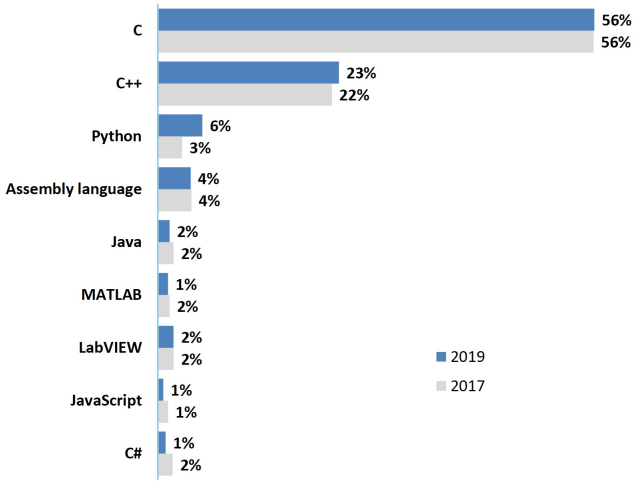

Figure 1.2: EETimes 2019 Embedded Markets Study [7]

To this day, the C language family continues to maintain its dominance in the embedded systems industry. The second-most popular language, C++, very often uses a highly specialised subset of the modern C++ standards. These subsets ban several high-level features of C++ and constrain the language, effectively making it behave more like C.

Notable in both surveys is the presence of modelling environments like MATLAB and LabVIEW. These frameworks are broadly used for designing entire systems that comprise multiple components. For instance, LabVIEW uses a data flow-driven programming model to connect several components in

[1]Note that the EETimes survey, unlike the VDC survey, doesn't allow multiple responses.

a system. However, the individual components are often configured to generate C programs, which constitute the heart of the systems.

This omnipresence of C is primarily for the reason that a lot of the legacy microcontroller vendors supported C compilers. This has reached a point today such that any new microcontroller that gets introduced into the market compulsorily supports a C compiler.

One of the key benefit of C is that it is a small and sufficiently low-level language that can enable the programmer to write resource-conscious programs. Restricted subsets of C, such as MISRA C [8], enable a programmer to write deterministic programs with statically predictable object lifetimes.

However, this strength of C as a “low-level systems language” can become a disadvantage in terms of high cost of software development. C is a memory-unsafe language, and this has had high costs on systems for the last several decades. According to the 2021 Common Weakness Enumeration (CWE) rankings by MITRE [9], out-of-bound writes remains the top vulnerability in software systems. For that specific CWE, we find the C, C++ and assembly languages as the most applicable platforms.

Naturally, in other software domains, where resource sensitivity is not a concern, there has been widescale migration to memory-safe languages like Java and Python. Now, despite the strong foothold of C in the resource-sensitive embedded systems space, we can compare between Fig 1.1 and Fig 1.2 to see Python’s growing popularity, overtaking Assembly and Java.

Python’s dynamic semantics is, in general, highly unsuitable for embedded systems. However, the popularity and syntactic familiarity of the language has resulted in a Python implementation - Micropython [10], which is gaining traction in the embedded systems space. Micropython implements the Python language with some minor differences from the reference implementation CPython, such as a compact representation of integers, restrictions on Python standard libraries, etc.

Although Python guarantees memory safety over C, it lacks in terms of expressing concurrent programs. The Python Language Reference defines a single-threaded language and the reference implementation CPython holds a Global Interpreter Lock (GIL) that prevents true multithreading. The language also lacks any fundamental support for real-time computations. Despite these limitations, the Micropython runtime provides a more resource-sensitive implementation compared to CPython, which perhaps explains the steady adoption of the language in the embedded space.

Having observed the industrial trends among embedded systems languages, we shall now turn our focus on research-oriented languages in this space.

1.1.2 Research Languages

The pervasiveness of embedded systems and the age of the research field has resulted in diverse strands of research on languages, frameworks and tooling infrastructure for embedded systems. Instead of an exhaustive literature survey, we shall selectively look at some of the past influential lines of work and an emerging programming model that could potentially impact the field.

Synchronous Languages

One of the most successful lines of research on embedded systems language is the synchronous language family. The most influential languages from this family are the three French languages - Esterel [11], Lustre [12], and Signal [13]. They are all based on a fundamental *synchrony hypothesis* that states -

> All reactions are assumed to be *instantaneous* - and therefore atomic in any possible sense.

The above essentially imposes a logical notion of time where all operations such as instruction-sequencing, inter-process communication, data handling, etc., happen instantaneously, *taking no time.* In practical implementations, the synchronous hypothesis is approximated to the assumption - a program can react to an external event before any further event occurs. The occurrence of an external event amounts to a clock tick in the logical clock.

The synchrony hypothesis is quite useful in the context of real-time systems to eliminate any *jitter* from the reaction time of a program. To realise the hypothesis in practice, there has been a long history of research on various compilation techniques for Esterel [14], which has influenced the other synchronous language implementations as well. These techniques have enabled synchronous languages to produce programs that occupy bounded memory.

The synchronous languages, however, do not aim to target all classes of reactive embedded systems. As discussed by de Simone et al. [15], "the focus of synchronous languages is to allow modeling and programming of systems where cycle (comptation step) precision is needed". Cycle precision of embedded systems can be found in areas such as hardware (clock cycles) and avionics. However, several classes of applications, like IoT, do not have a regular, periodic clock that drives external events. The logical clock ticks for several such systems are sparsely spread. There has been recent work on the sparse synchronous model [16] to address such systems in the synchronous framework.

Additionally, synchronous languages often do not support general syntactic constructs of a language. For instance, Esterel divides a reactive program into three parts - (i) the I/O-interfacing layer, (ii) the reactive kernel and (iii) data-handling layer [11]. Out of the three layers, Esterel is used to describe only the reactive kernel. The data handling, involving classical computations, is handled by some form of a host language, where Esterel is embedded. Similarly, the I/O-interfacing, such as interrupt-handling, reading/writing of data, etc., which constitutes a large part of a reactive program, has to be designed entirely in a host language, very likely C in the case of embedded systems.

Giotto

Closely related to the synchronous family of languages is the time-triggered hard real-time language - Giotto [17]. Giotto draws inspiration from the time-triggered architecture (TTA) [18] that found application in safety-critical systems. In contrast with event-triggered (or event-driven) systems, time-triggered systems like Giotto operate solely according to a pre-determined and set task schedule.

Giotto operates under what it calls *fixed logical execution time* (FLET) assumption, which states -

> The execution times associated with all computation and communication activities are fixed and determined by the model, not the platform. In Giotto, the logical execution time of a task is always exactly the period of the task, and the logical execution times of all other activities are always zero.

The above differs from the synchrony hypothesis in the sense that it is a formally weaker notion of value propagation (zero delay vs unit delay). The implication of this difference affects the compilation process of the respective languages; whereas, in the compilation of synchronous languages, the focus is on fixed-point analysis; in the case of Giotto, the importance is on schedulability analysis [19]. Accordingly, Giotto abstracts away its scheduling process to a separate virtual machine called the Embedded Machine [20].

Giotto, as well as the synchronous languages, target the same category of embedded applications - real-time control applications with a periodic *heartbeat*. Likewise, both programming models are ill-suited for applications with a sparse and *aperiodic* control pulse.

Timber

The Timber programming language [21] was an attempt to design a high-level language targeting embedded devices. Timber wanted to bring the object-oriented programming paradigm to the embedded systems space. One of the advantages of an object-oriented language - a principled state management discipline - would be an improvement over the low-level practices such as state-transition tables, being used in C. Although there exist other real-time, object-oriented languages like RTSJ [22], they heavily restrict any form of dynamic object behaviour.

Timber features a rigorous type system and formal semantics to enable the construction of deterministic programs. The semantics of Timber operates under a central property -

> Each reaction will terminate independent of further events, hence a system described in Timber will be responsive at all times (under the limitation of available CPU resources) and free of deadlocks.

The above property serves as the foundation for further system analyses on Timber. The language, additionally, guarantees mutual exclusion over any form of concurrent state-accesses within an application through its syntactic constructs. Timber also provides a rich set of temporal primitives, such as *before*, *after*, etc., for specifying the absolute time at which a particular reaction should occur.

The major hurdle in realising the Timber project was designing a language runtime that could fully support dynamic object-oriented behaviour in a resource-constrained scenario. Inevitably, this challenge requires the design of advanced garbage collection mechanisms that can support seamless pausing and incremental collection techniques. There are brief allusions to an experimental *interruptible* reference-counting collector in the Timber technical report [23], of which we could not find any peer-reviewed publication. There has, however, been a theoretical description of an incremental garbage collector [24] and its scheduling for real-time systems [25] from the team behind Timber. The development of the project, based on postings to the Timber mailing list, seems to have terminated around 2009-10.

Embedded Domain-Specific Languages

A well-studied line of research is designing restricted languages for specific domains, such as embedded systems, and then, instead of constructing a full-fledged compiler and runtime, utilising the compiler infrastructure of a general-purpose language. Such languages are termed embedded domain-specific languages (EDSLs) [26] and have been used to program a variety of embedded applications.

The Copilot EDSL [27] has been used to design hard real-time runtime monitoring tools. It is a stream-based dataflow language that can generate small constant-time and constant-space C programs, implementing embedded monitors. Another example of an embedded-system EDSL is Ivory [28], which enables writing memory-safe C programs.

The advantage of EDSLs is that they bring high-level abstraction from the host language, such as Haskell, to the programming of fairly low-level applications. Additionally, EDSLs have very little runtime overheads compared to a high-level language that is natively run on embedded systems.

However, programming with an EDSL often involves learning two sets of semantics - one of the host language and the other of the EDSL itself. Frequently, the two sets of semantics oppose each other, and the resultant program is challenging to maintain and understand. Hickey et al. [29] discuss the lessons learned in programming an embedded systems application using an EDSL and cite challenges such as illegible type-error messages and undefined C-program generation.

Functional Reactive Programming

The programming model of *Functional Reactive Programming (FRP)* was born to declaratively express graphical programs [30]. A fundamental difference of this programming model from the synchronous model is the notion of *continuous-time* rather than discrete. The notion of continuous-time is useful for declaratively expressing such things as mouse movement in graphical applications.

FRP models continuous objects through an abstraction it calls *Behaviour*. It also provides an abstraction called *Event* to model discrete operations such as mouse-clicks. Along with behaviours and events FRP provides a series of higher-order functions and combinators to describe reactive graphical applications.

Although initially limited to graphics, the programming model was found suitable for describing modern web applications via popular Javascript libraries like React [31]. Theoretically, the reactive nature of the model seems to be a good fit for embedded systems.

However, one of the most challenging aspects of FRP has been realising the model of continuous-time in practical implementations. Most original implementations of FRP suffered from severe memory leaks. There has been work [32] to fix these leaks but often at the cost of expressivity of the model.

Another challenge of FRP is that it is often embedded[2] within a general-purpose language like Haskell, which is not exclusively designed for highly reactive applications. As a result, there is a lot of unnecessary plumbing required to make the internal FRP model interact with the external environment through the I/O monad [33].

In general, there is a lot of proliferation in the variety of APIs and implementations available for FRP. There are studies [34] to analyse the strengths and weaknesses of the different design choices. There has also been experiments on distributing an FRP runtime across multiple devices [35]. From the perspective of embedded systems, there has been past research on highly-restricted FRP implementations targeting resource-constrained PIC microcontrollers [36].

Overall, the FRP programming model has potential application for embedded systems, but there is still no consensus on the ideal API or its implementation. There needs to be more research on designing resource-sensitive FRP runtimes as well as on bringing time-critical computations to FRP.

1.2 The Gap

Our survey, in the last section, has shown the industry trends as well as research activities related to embedded systems languages. At this point, we would like to remind the reader about the critical properties that are essential for designing an embedded-systems language in the tables below -

Reactivity	Resource-sensitivity
1. I/O-bound	1. Low power
2. Concurrent	2. Low memory
3. Timing-aware	3. Low clock-cycles

While the C programming language's closeness to the hardware allows a C programmer to write resource-sensitive programs, it is plagued with security vulnerabilities discussed earlier. Additionally, C is not a concurrent language. There are some ad-hoc libraries such as Protothreads [37] to mimic concurrent behaviour, but the intrinsic language semantics is not concurrent. There is a gap for a high-level language that embodies the reactive properties shown above while running programs in a resource-sensitive manner.

The research languages that we have discussed all attempt to address this gap. However, they fall short in certain areas. For instance, the synchronous languages and Giotto are well designed for timing-critical applications but exclusively target periodic applications with a regular heartbeat. Additionally, the I/O-handling, which constitutes a major part of typical embedded systems applications, is excluded from the programming models.

Finally, most other high-level programming abstractions, such as objects in Timber or Functional Reactive Programming, lack resource-sensitive implementations. Given this state of the ecosystem, we attempt to address the following research question -

[2]refers to the *embedding* of a language, not to be confused with embedded systems

Research Question - *What are the fundamental high-level abstractions that address the reactive nature of embedded systems, and how should these abstractions be implemented in a resource-sensitive manner?*

1.3 Contributions

The hope, through our contributions, is to present high-level abstractions that could have resource-sensitive implementations. Our contributions are broadly divided into two parts. The first part comprises a paper that describes an experimental reactive programming language. The second part contains two papers describing the design of a high-level API and its runtime implementation for embedded systems. We briefly summarise them in the following section -

Part I

1.3.1 Paper A : Hailstorm

The first paper presents Hailstorm [38], a statically-typed, purely-functional, reactive domain-specific language. It uses the Arrowized FRP [39] formulation of FRP to program embedded devices. The most central type in the language is that of a signal function, $SF\ a\ b$, where a and b denote polymorphic type variables. Signal functions are representations of a dataflow from type a to b.

We further extended this representation with the concept of a resource type [40]. A resource type is type-level label that can be used to uniquely identify various external resources. The new type of a signal function becomes $SF\ r\ a\ b$, where r denote a polymorphic resource label. For instance, two sensors that can supply an Int and $Float$ value type respectively, will have the following types in Hailstorm -

```
resource S1
resource S2

sensor1 :: SF S1 () Int
sensor2 :: SF S2 () Float
```

The unit type - () - above indicates that the sensor interacts with the external world. Hailstorm, additionally, provides a family of combinators to declaratively compose the data flowing through the various signal functions -

```
mapSignal# : (a -> b) -> SF Empty a b
(>>>) : SF r_1 a b -> SF r_2 b c -> SF (r_1 ∪ r_2) a c
(&&&) : SF r_1 a b -> SF r_2 a c -> SF (r_1 ∪ r_2) a (b, c)
(***) : SF r_1 a b -> SF r_2 c d -> SF (r_1 ∪ r_2) (a, c) (b, d)
```

The technical details about the type-level union and its semantics are described in-depth in the paper. As discussed earlier, FRP models are often embedded within a host language, which make any form of interaction with the external world quite syntactically awkward. The introduction of resource types is done to resolve this issue, and we detail, using examples, in the paper on how a resource label can allow the *correct* composition of signal functions.

The Hailstorm language has an LLVM and Erlang backend. The Erlang backend, in particular, was used to prototype experiments on the GRiSP microcontroller boards. Our evaluations consisted of writing very small prototype applications in Hailstorm like a watchdog process, a simplified traffic light system and a railway level-crossing simulator. One of the complications associated with resource types is the linear increase in the number of type labels as the number of resources (i.e. external inputs) increases, which often overshadow the type-signatures within the programs.

We also carried out micro-benchmarks on the memory consumption and response time of the programs. The memory footprint of the Hailstorm programs was in the order of 2-3 MB, owing to the size of the Erlang runtime. The response time of the programs was in the range of 100-150 microseconds.

Hailstorm was primarily an experiment to address the I/O-bound nature of embedded applications. Concurrency in FRP is implicit, and as for timing, the language lacked any form of real-time APIs. Additionally, the code generated from Hailstorm is polling-based, which is typically energy-inefficient compared to event-based programs.

To remedy the above shortcomings, we realised that Hailstorm or any reactive programming model, in general, are too high-level. There is a missing layer of abstraction in between, as shown in Fig. 1.3, something akin to a language runtime.

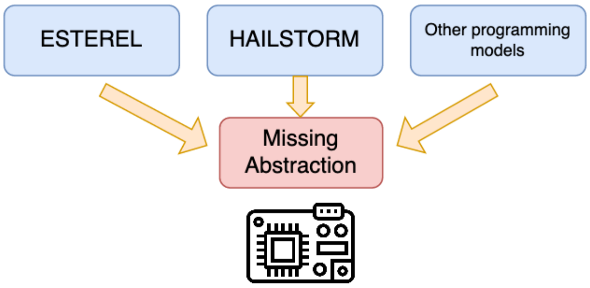

Figure 1.3: A missing layer of abstraction

The expectation of this layer is to be more low-level such that it can handle the complexities of callback-based driver interfaces. It should naturally feature explicit concurrency to capture the concurrent nature of the hardware while including some notion of timing. Additionally, it should provide a resource-sensitive runtime for supporting various programming models. The next part discusses our attempt at such a project.

Part II

Our discussions till now have concentrated on programming languages for embedded systems. Now, as we descend into a lower layer of abstraction, we shall briefly survey some low-level frameworks, such as virtual machines, targeting embedded devices.

Low-level tools for Embedded Systems

Virtual Machines

There have been attempts at porting popular general-purpose languages to run on microcontrollers using virtual machines. Some examples are OMicroB [41] supporting OCaml, Picobit [42] supporting Scheme and AtomVM [43] supporting Erlang. In the real-time space, a safety-critical VM that can provide hard real-time guarantees on Real-Time Java programs is the FijiVM [44] implementation. The FijiVM project invented the Schism garbage collector [45], a concurrent garbage collector that can handle real-time applications on multicore embedded devices.

WebAssembly Micro Runtime

The WebAssembly project (WASM) has spawned sub-projects like WebAssembly Micro Runtime (WAMR) [46] that allows running WASM-supported languages on microcontrollers. Language like JavaScript and Rust currently have work-in-progress and experimental backends supporting WebAssembly. Reliable benchmarks on how those languages and their runtimes perform on microcontrollers are still not available, and research on optimizing them specifically for microcontrollers is still at a nascent stage.

Operating Systems

The last layer of abstraction that lies before the hardware is the operating system. Typically in the context of embedded systems, these are often specialised with real-time APIs and are called *real-time* operating systems (RTOS). Examples of popular RTOSes are Zephyr OS [47], ChibiOS [48] and FreeRTOS [49].

Generally, RTOSes are much lighter than desktop operating systems. They typically come equipped with a scheduler, kernel services and something called a hardware abstraction layer (HAL). A HAL serves as an abstraction for accessing various microcontroller peripherals like GPIO, ADC, SPI and so on and also take care of clock-related and board-level initialization.

Apart from the core services discussed above, RTOSes vary in terms of higher levels of abstractions offered. FreeRTOS is more bare-bones compared to something like Zephyr OS, which provides network drivers, implementations of high-level networking protocols, etc. The programming language supported by all of the above RTOSes is exclusively C.

1.3.2 Papers B & C: Synchron

Our current project, Synchron, attempts to fill the missing abstraction of Fig. 1.3 such that high-level programming models, like FRP, can be hosted more naturally on embedded devices. Synchron is a specialised runtime API designed for expressing (i) I/O-bound, (ii) concurrent and (iii) timing-aware programs. The architecture of Synchron consists of three parts -

- Runtime - The principal component of Synchron is a specialised runtime consisting of nine built-in operations and a scheduler. The runtime allows the creation of concurrent user-level processes (green threads) and provides operators for declaratively expressing interactions between the software processes and hardware interrupts. The power-efficient scheduling of the processes is managed by the Synchron scheduler.

- Low-level Bridge - The Synchron runtime interacts with the various hardware drivers through a low-level *bridge* interface. The interface is general enough such that it can be implemented by both synchronous drivers (like LED) as well as asynchronous drivers (like UART).

- Underlying OS - The Synchron runtime is run atop an underlying RTOS such as ZephyrOS or ChibiOS. The OS supplies the actual hardware drivers that implements the low-level bridge interface described above. We have designed our runtime interfaces in a modular fashion such that other operating systems, such as FreeRTOS, can be easily plugged in.

Our implementation of Synchron is in the form of a bytecode-interpreted virtual machine called the SynchronVM. The execution engine of SynchronVM is based on the Categorical Abstract Machine [50], which supports the cheap creation of closures to support functional programming languages. Fig. 1.4 below provides a graphical description of the architecture of Synchron.

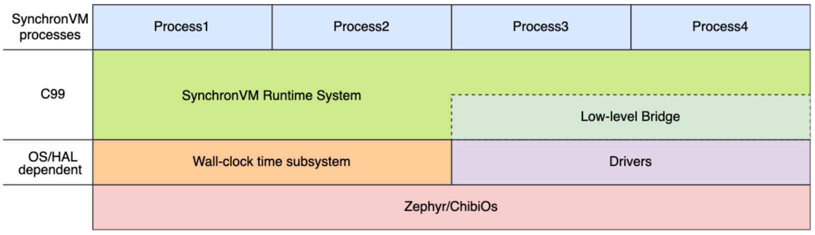

Figure 1.4: Architecture of Synchron

The Synchron API

The core API of Synchron comprises of nine functions. We show the type-signatures of those functions in Fig. 1.5 below.

```
spawn   : (() -> ()) -> ThreadId
channel : ()  -> Channel a
send    : Channel a -> a -> Event ()
recv    : Channel a -> Event a
choose  : Event a   -> Event a  -> Event a
wrap    : Event a   -> (a -> b) -> Event b
sync    : Event a   -> a
syncT   : Time -> Time -> Event a -> a
spawnExternal : Channel a -> Driver -> ExternalThreadId
```

Figure 1.5: The Synchron API

The Synchron API is derived from Concurrent ML (CML) [51], which is a synchronous message-passing–based concurrency model. The fundamental difference of CML from standard synchronous message-passing models, such as communicating sequential processes [52], is the separation between the *intent* and *act* of communication. This separation is captured by first-class values called *Events*.

An event is an abstraction to represent deferred communication. In contrast with a rudimentary protocol involving single message sends and receives, the CML combinators such as `wrap` and `choose` can compose elaborate communication protocols involving multiple sends and receives.

Our extension of the CML API involves the last two functions in Fig. 1.5 - `syncT` and `spawnExternal`. The first extension, `syncT`, allows a programmer to specify the exact timing window at which an event synchronisation should happen. The first argument to `syncT` represents the baseline of the operation, while the second argument is the deadline. The `syncT` operator provides an opportunity to dynamically prioritise concurrent timed processes instead of static-priority APIs provided by typical RTOSes.

The second extension, `spawnExternal`, models the external hardware drivers as processes themselves. Modelling the drivers as processes allows a programmer to apply the entire message-passing API to low-level drivers interactions such as interrupt-handling. The serialisation and deserialisation between software messages and hardware interrupts are handled by the runtime.

We describe our design and implementation of the SynchronVM in the form of two papers [53, 54]. Do note that in the first of the two papers, we call the VM "SenseVM". However, we later renamed the VM, keeping in mind the synchronous nature of our API, to SynchronVM.

SynchronVM currently supports the nRF52840DK and the STM32F4 microcontroller boards. We carried out our evaluations of the SynchronVM with the help of a musical application, which involves some soft real-time components. Other micro-benchmarks were carried out on response times, memory usage and power consumption.

Our preliminary results are encouraging and show that in terms of power usage, a program running on SynchronVM has the same amount of momentary power consumption as a C program written using callback registration. Of course, if the power usage is integrated over time, a C program would be more power-efficient. However, the trade-off of programming with high-level abstractions is an attractive proposition.

In terms of memory usage, a SynchronVM program occupies tens to hundreds of kilobytes, which is beneficial for memory constrained microcontrollers. The response times of our benchmarks are typically 2-3x times slower than the C equivalents. A point to be noted here is that our execution engine is based on the categorical abstract machine, which is known to be four times slower, on average, than the Zinc abstract machine [55]. Additionally, we use a basic stop-the-world, non-moving, mark-and-sweep garbage collector, which contributes to the slowness of the response times.

Overall, SynchronVM provides a possible answer to the research question that we posed earlier. We have identified a set of high-level abstractions for writing concurrent, I/O-bound and timing-aware embedded system programs. We further provide a resource-sensitive implementation of the proposed abstractions and benchmark our implementation. Our initial results on memory and power usage look promising, while response times could be further improved.

There are several open avenues for optimisations, such as moving to a faster execution engine, switching to a register-based VM, performing *ahead-of-time* compilation and using advanced generational garbage collectors. Aside from the various optimisation opportunities, in the next section, we discuss the more research-oriented challenges that we hope to address in the future.

1.4 Future Work

1.4.1 Region-based Memory Management

One of the most critical challenges in hosting a high-level programming language on embedded systems is the aspect of dynamic memory management. Object-oriented as well as functional programming paradigms feature plenty of dynamic memory allocation. Typically managed runtimes employ garbage collectors to deallocate the dynamically allocated memory. However, the points of time where the memory will be deallocated by the garbage collector or the total time required for deallocation is unpredictable. This unpredictability complicates the scheduling of real-time applications (especially hard real-time) on embedded systems [56].

A promising line of work to mitigate the non-determinism discussed above is the region-based memory management (RBMM) discipline [57]. RBMM arranges the memory into a stack of regions that gets deallocated in a last-in-first-out fashion. The principal component of RBMM is a series of type-based static analysis passes, called region inference [58], that conservatively determines the scope of a piece of dynamic memory. Accordingly, it then allocates the memory into the ideal position within the region stack identified by the analysis.

The advantage of RBMM is that memory deallocation happens in constant time. Additionally, the structure of the program provides insights into the scoping pattern that dictates the lifetime of the various memory allocations. It has the potential to be used in embedded systems applications as a lot of typical C programs involve identifying the same static arrangement of the memory layout, which RBMM can infer automatically.

However, the authors and implementers of RBMM have identified weaknesses of the approach that often lead to memory leaks in practical programs [59]. In most cases, RBMM has to be used in conjunction with a garbage collector, which reintroduces the same unpredictability that real-time applications preferably avoid. Also, region-based memory leaks were found to be notoriously hard to detect and then debug, which often involved redefining the program structure.

A recent study [60] has analysed the behaviour of region-based, forever-alive server programs and found the weakness to be that the region-inference algorithm conservatively places all of the memory into a global infinite region, which continues growing till memory leaks. A possible research track would be identifying the root cause behind this conservative estimation of the region-inference algorithm and employing more optimistic region-allocation policies, which could roll back the allocation by dynamically moving the data across regions. Also, recent research on integrating generational garbage collection with RBMM [61] is a prospective memory management strategy for SynchronVM.

1.4.2 Security of Embedded Systems

One of the most vital areas of research interest is the security of embedded systems. As discussed in the earlier sections, low-level, memory-unsafe languages such as C and C++ introduce a vast array of memory-related vulnerabilities. The introduction of high-level, managed programming languages aims to reduce a large portion of such security weaknesses.

However, there exists a large class of hardware-based vulnerabilities [62] as well as communication protocol threats [63], which cannot be prevented by simply using high-level languages. Their mitigation requires advanced security techniques such as specialised hardware support.

ARM microcontrollers (Cortex-M), which universally dominates the embedded systems market, provides a special security mode called the ARM TrustZone [64]. There has been a recent surge of interest from the security community in TrustZone [65], which allows isolating critical security firmware, assets and private information. Aside from isolation, TrustZone also provides building blocks to implement end-to-end security solutions, namely, trusted I/O paths, secure storage, and remote attestation.

Accessing the TrustZone features require interactions with very low-level C and assembly APIs. An improvement here would be using the ARM TrustZone API and constructing secure application compartments on SynchronVM. Consequently, applications hosted on the SynchronVM, using any host language, could isolate their security-sensitive logic from the rest of the application. There has been a related project in this field for desktop applications running on the .NET language runtime called the Trusted Language Runtime [66].

CHERI

An exciting new development has been the CHERI capability model [67], which extends the RISC instruction set with capability-based memory protection to mitigate common memory vulnerabilities. The very recent release of the ARM Morello board [68], which integrates the CHERI capability model into actual physical hardware, provides an excellent opportunity to experiment with building higher-level platforms such as operating systems and virtual machines that test the utility of the CHERI model.

A research avenue comparable with the ARM TrustZone–based secure compartments would be using the CHERI capabilities for compartmentalisation in the SynchronVM. There has been recent work to port the C/C++–based Boehm-Demers-Weiser garbage collector to CHERI hardware [69], which could serve as an experimental memory manager to support the SynchronVM (written entirely in C99) on such devices.

Bibliography

[1] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 1–15. [Online]. Available: https://doi.org/10.1007/11813040_1

[2] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*, ser. NATO ASI Series, K. R. Apt, Ed., vol. 13. Springer, 1984, pp. 477–498. [Online]. Available: https://doi.org/10.1007/978-3-642-82453-1_17

[3] S. Ravi, A. Raghunathan, P. C. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, 2004. [Online]. Available: https://doi.org/10.1145/1015047.1015049

[4] Z. Zhang, M. C. Y. Cho, C. Wang, C. Hsu, C. K. Chen, and S. Shieh, "Iot security: Ongoing challenges and research opportunities," in *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*. IEEE Computer Society, 2014, pp. 230–234. [Online]. Available: https://doi.org/10.1109/SOCA.2014.58

[5] A. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial internet of things," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, pp. 54:1–54:6. [Online]. Available: https://doi.org/10.1145/2744769.2747942

[6] VDCResearchSurvey. (2011) Embedded engineer survey results. [Online]. Available: https://blog.vdcresearch.com/embedded_sw/2011/06/2011-embedded-engineer-survey-results-programming-languages-used-to-develop-software.html

[7] EETimes. (2019) 2019 embedded markets study. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[8] L. Hatton, “Safer language subsets: an overview and a case history, MISRA C,” *Inf. Softw. Technol.*, vol. 46, no. 7, pp. 465–472, 2004. [Online]. Available: https://doi.org/10.1016/j.infsof.2003.09.016

[9] M. Corp. (2021) 2021 cwe top 25 most dangerous software errors. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[10] D. George. (2014) Micropython. [Online]. Available: https://micropython.org/

[11] G. Berry and G. Gonthier, “The esterel synchronous programming language: Design, semantics, implementation,” *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: https://doi.org/10.1016/0167-6423(92)90005-V

[12] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, “Lustre: A declarative language for programming synchronous systems,” in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987.* ACM Press, 1987, pp. 178–188. [Online]. Available: https://doi.org/10.1145/41625.41641

[13] T. Gautier and P. Le Guernic, “SIGNAL: A declarative language for synchronous programming of real-time systems,” in *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 274. Springer, 1987, pp. 257–277. [Online]. Available: https://doi.org/10.1007/3-540-18317-5_15

[14] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel.* Springer, 2007. [Online]. Available: https://doi.org/10.1007/978-0-387-70628-3

[15] R. de Simone, J. Talpin, and D. Potop-Butucaru, “The synchronous hypothesis and synchronous languages,” in *Embedded Systems Handbook*, R. Zurawski, Ed. CRC Press, 2005. [Online]. Available: https://doi.org/10.1201/9781420038163.ch8

[16] S. A. Edwards and J. Hui, “The sparse synchronous model,” in *Forum for Specification and Design Languages, FDL 2020, Kiel, Germany, September 15-17, 2020.* IEEE, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FDL50818.2020.9232938

[17] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming,” in *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, ser. Lecture Notes in Computer Science, T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211. Springer, 2001, pp. 166–184. [Online]. Available: https://doi.org/10.1007/3-540-45449-7_12

[18] H. Kopetz, *Real-time systems - design principles for distributed embedded applications*, ser. The Kluwer international series in engineering and computer science. Kluwer, 1997, vol. 395.

[19] C. M. Kirsch, “Principles of real-time programming,” in *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. L. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Springer, 2002, pp. 61–75. [Online]. Available: https://doi.org/10.1007/3-540-45828-X_6

[20] T. A. Henzinger and C. M. Kirsch, “The embedded machine: Predictable, portable real-time code,” in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 315–326. [Online]. Available: https://doi.org/10.1145/512529.512567

[21] M. Carlsson, J. Nordlander, and D. Kieburtz, “The semantic layers of timber,” in *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, ser. Lecture Notes in Computer Science, A. Ohori, Ed., vol. 2895. Springer, 2003, pp. 339–356. [Online]. Available: https://doi.org/10.1007/978-3-540-40018-9_22

[22] JCP. (2001) The real-time specification for java. [Online]. Available: https://www.rtsj.org/

[23] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, *Time for timber*. Luleå tekniska universitet, 2005.

[24] M. Kero, J. Nordlander, and P. Lindgren, “A correct and useful incremental copying garbage collector,” in *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, G. Morrisett and M. Sagiv, Eds. ACM, 2007, pp. 129–140. [Online]. Available: https://doi.org/10.1145/1296907.1296924

[25] M. Kero and S. Aittamaa, “Scheduling garbage collection in real-time systems,” in *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, part of ESWeek ’10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010*, T. Givargis and A. Donlin, Eds. ACM, 2010, pp. 51–60. [Online]. Available: https://doi.org/10.1145/1878961.1878971

[26] P. Hudak, “Building domain-specific embedded languages,” *ACM Comput. Surv.*, vol. 28, no. 4es, p. 196, 1996. [Online]. Available: https://doi.org/10.1145/242224.242477

[27] L. Pike, A. Goodloe, R. Morisset, and S. Niller, “Copilot: A hard real-time runtime monitor,” in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 345–359. [Online]. Available: https://doi.org/10.1007/978-3-642-16612-9_26

[28] T. Elliott, L. Pike, S. Winwood, P. C. Hickey, J. Bielman, J. Sharp, E. L. Seidel, and J. Launchbury, "Guilt free ivory," in *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, B. Lippmeier, Ed. ACM, 2015, pp. 189–200. [Online]. Available: https://doi.org/10.1145/2804302.2804318

[29] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury, "Building embedded systems with embedded dsls," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 3–9. [Online]. Available: https://doi.org/10.1145/2628136.2628146

[30] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, S. L. P. Jones, M. Tofte, and A. M. Berman, Eds. ACM, 1997, pp. 263–273. [Online]. Available: https://doi.org/10.1145/258948.258973

[31] J. Walke. (2013) React.js. [Online]. Available: https://reactjs.org/

[32] H. Liu and P. Hudak, "Plugging a space leak with an arrow," *Electron. Notes Theor. Comput. Sci.*, vol. 193, pp. 29–45, 2007. [Online]. Available: https://doi.org/10.1016/j.entcs.2007.10.006

[33] A. van der Ploeg and K. Claessen, "Practical principled FRP: forget the past, change the future, frpnow!" in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, K. Fisher and J. H. Reppy, Eds. ACM, 2015, pp. 302–314. [Online]. Available: https://doi.org/10.1145/2784731.2784752

[34] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, 2013. [Online]. Available: https://doi.org/10.1145/2501654.2501666

[35] K. Shibanai and T. Watanabe, "Distributed functional reactive programming on actor-based runtime," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*, J. D. Koster, F. Bergenti, and J. Franco, Eds. ACM, 2018, pp. 13–22. [Online]. Available: https://doi.org/10.1145/3281366.3281370

[36] Z. Wan, W. Taha, and P. Hudak, "Event-driven FRP," in *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, ser. Lecture Notes in Computer Science, S. Krishnamurthi and C. R. Ramakrishnan, Eds., vol. 2257. Springer, 2002, pp. 155–172. [Online]. Available: https://doi.org/10.1007/3-540-45587-6_11

[37] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, A. T. Campbell, P. Bonnet, and J. S. Heidemann, Eds. ACM, 2006, pp. 29–42. [Online]. Available: https://doi.org/10.1145/1182807.1182811

[38] A. Sarkar and M. Sheeran, "Hailstorm: A statically-typed, purely functional language for iot applications," in *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020.* ACM, 2020, pp. 12:1–12:16. [Online]. Available: https://doi.org/10.1145/3414080.3414092

[39] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002, pp. 51–64.

[40] D. Winograd-Cort, H. Liu, and P. Hudak, "Virtualizing real-world objects in FRP," in *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, ser. Lecture Notes in Computer Science, C. V. Russo and N. Zhou, Eds., vol. 7149. Springer, 2012, pp. 227–241. [Online]. Available: https://doi.org/10.1007/978-3-642-27694-1_17

[41] S. Varoumas, B. Vaugon, and E. Chailloux, "A generic virtual machine approach for programming microcontrollers: the omicrob project," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

[42] V. St-Amour and M. Feeley, "PICOBIT: A compact scheme system for microcontrollers," in *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. T. Morazán and S. Scholz, Eds., vol. 6041. Springer, 2009, pp. 1–17. [Online]. Available: https://doi.org/10.1007/978-3-642-16478-1_1

[43] D. Bettio. (2017) Atomvm. [Online]. Available: https://github.com/bettio/AtomVM

[44] F. Pizlo, L. Ziarek, and J. Vitek, "Real time java on resource-constrained platforms with fiji VM," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ser. ACM International Conference Proceeding Series, M. T. Higuera-Toledano and M. Schoeberl, Eds. ACM, 2009, pp. 110–119. [Online]. Available: https://doi.org/10.1145/1620405.1620421

[45] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek, "Schism: fragmentation-tolerant real-time garbage collection," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming*

Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010, B. G. Zorn and A. Aiken, Eds. ACM, 2010, pp. 146–159. [Online]. Available: https://doi.org/10.1145/1806596.1806615

[46] (2019) Wamr - webassembly micro runtime. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[47] T. L. Foundation, "Zephyr RTOS," https://www.zephyrproject.org/, accessed 2021-11-28.

[48] G. D. Sirio, "ChibiOS," https://www.chibios.org/dokuwiki/doku.php, accessed 2021-11-28.

[49] R. Barry, "FreeRTOS," https://www.freertos.org/, accessed 2021-11-28.

[50] G. Cousineau, P. Curien, and M. Mauny, "The categorical abstract machine," in *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, ser. Lecture Notes in Computer Science, J. Jouannaud, Ed., vol. 201. Springer, 1985, pp. 50–64. [Online]. Available: https://doi.org/10.1007/3-540-15975-4_29

[51] J. H. Reppy, "Concurrent ML: design, application and semantics," in *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, ser. Lecture Notes in Computer Science, P. E. Lauer, Ed., vol. 693. Springer, 1993, pp. 165–198. [Online]. Available: https://doi.org/10.1007/3-540-56883-2_10

[52] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[53] A. Sarkar, R. Krook, B. J. Svensson, and M. Sheeran, "Higher-order concurrency for microcontrollers," in *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, H. Kuchen and J. Singer, Eds. ACM, 2021, pp. 26–35. [Online]. Available: https://doi.org/10.1145/3475738.3480716

[54] A. Sarkar, B. J. Svensson, and M. Sheeran, "Synchon - an api for embedded systems," 2022, under submission.

[55] X. Leroy, "The zinc experiment: an economical implementation of the ml language," Ph.D. dissertation, INRIA, 1990.

[56] K. Hammond, "Is it time for real-time functional programming?" in *Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, United Kingdom, 11-12 September 2003*, ser. Trends in Functional Programming, S. Gilmore, Ed., vol. 4. Intellect, 2003, pp. 1–18.

[57] M. Tofte and J. Talpin, “Region-based memory management,” *Inf. Comput.*, vol. 132, no. 2, pp. 109–176, 1997. [Online]. Available: https://doi.org/10.1006/inco.1996.2613

[58] M. Tofte and L. Birkedal, “A region inference algorithm,” *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 724–767, 1998. [Online]. Available: https://doi.org/10.1145/291891.291894

[59] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg, “A retrospective on region-based memory management,” *High. Order Symb. Comput.*, vol. 17, no. 3, pp. 245–265, 2004. [Online]. Available: https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[60] R. Krook, “Region-based memory management and actor model concurrency an initial study of how the combination performs,” 2020.

[61] M. Elsman and N. Hallenberg, “Integrating region memory management and tag-free generational garbage collection,” *J. Funct. Program.*, vol. 31, p. e4, 2021. [Online]. Available: https://doi.org/10.1017/S0956796821000010

[62] J. Obermaier and S. Tatschner, “Shedding too much light on a microcontroller’s firmware protection,” in *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, W. Enck and C. Mulliner, Eds. USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier

[63] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, “The matter of heartbleed,” in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, C. Williamson, A. Akella, and N. Taft, Eds. ACM, 2014, pp. 475–488. [Online]. Available: https://doi.org/10.1145/2663716.2663755

[64] ARM. (2014) Arm trustzone-m. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m

[65] S. Pinto and N. Santos, “Demystifying arm trustzone: A comprehensive survey,” *ACM Comput. Surv.*, vol. 51, no. 6, pp. 130:1–130:36, 2019. [Online]. Available: https://doi.org/10.1145/3291047

[66] N. Santos, H. Raj, S. Saroiu, and A. Wolman, “Using ARM trustzone to build a trusted language runtime for mobile applications,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 67–80. [Online]. Available: https://doi.org/10.1145/2541940.2541949

[67] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, “The CHERI capability model: Revisiting RISC in an age of risk,” in *ACM/IEEE 41st International Symposium on Computer*

Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. IEEE Computer Society, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853201

[68] ARM. (2022) Arm morello boards announcement. [Online]. Available: https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing

[69] D. Jacob and J. Singer, "Capability boehm: challenges and opportunities for garbage collection with capability hardware," in *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2022, pp. 81–87.