Thesis for The Degree of Licentiate of Engineering

Functional Programming for Embedded Systems

Abhiroop Sarkar

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2022

Functional Programming for Embedded Systems

Abhiroop Sarkar

ISBN xxx-xx-xxxx-xxx-x
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr xxxx.
ISSN 0346-718X

Technical Report No XXXX
Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2022.

"The price of reliability is the pursuit of the utmost simplicity."
- Tony Hoare

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

Acknowledgment

Acknowledgement page. Include also funding information here!

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

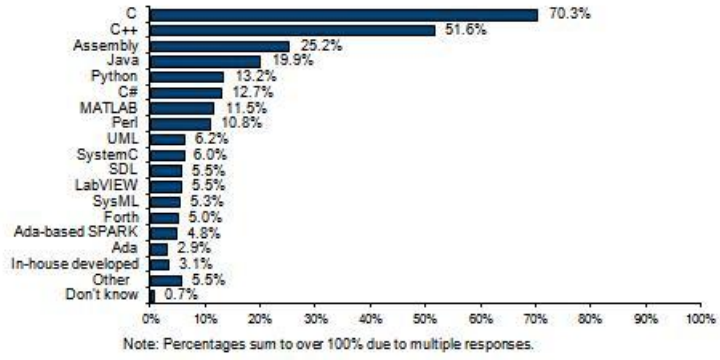


Figure 1.1: VDC 2011 Embedded Engineer Survey Results [6]

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].

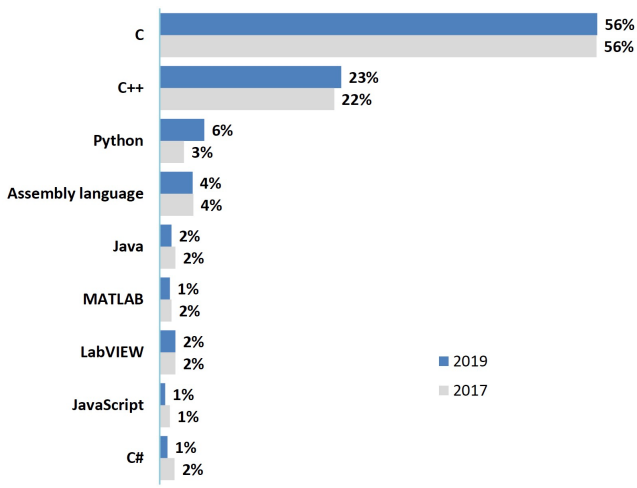


Figure 1.2: EETimes 2019 Embedded Markets Study [7]

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

Resource-sensitivity
1. Low power
2. Low memory
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 r1 a b -> SF r2 b c -> SF (r1 ∪ r2) a c
(&&&) : SF r1 a b -> SF r2 a c -> SF (r1 ∪ r2) a (b, c)
(***) : SF r1 a b -> SF r2 c d -> SF (r1 ∪ r2) (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.

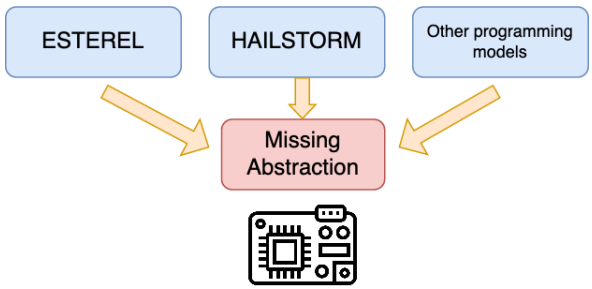


Figure 1.3: A missing layer of abstraction

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.

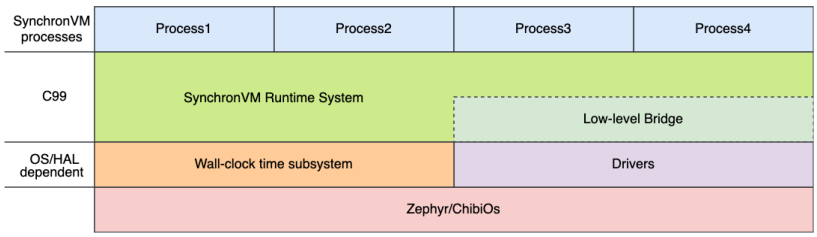


Figure 1.4: Architecture of Synchron

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 abstrac-

tions 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.

PART 1

Chapter 2

The Hailstorm IoT Language

Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications

Abhiroop Sarkar
sarkara@chalmers.se
Chalmers University
Gothenburg, Sweden

Mary Sheeran
mary.sheeran@chalmers.se
Chalmers University
Gothenburg, Sweden

ABSTRACT

With the growing ubiquity of *Internet of Things* (IoT), more complex logic is being programmed on resource-constrained IoT devices, almost exclusively using the C programming language. While C provides low-level control over memory, it lacks a number of high-level programming abstractions such as higher-order functions, polymorphism, strong static typing, memory safety, and automatic memory management.

We present Hailstorm, a statically-typed, purely functional programming language that attempts to address the above problem. It is a high-level programming language with a strict typing discipline. It supports features like higher-order functions, tail-recursion, and automatic memory management, to program IoT devices in a declarative manner. Applications running on these devices tend to be heavily dominated by I/O. Hailstorm tracks side effects like I/O in its type system using *resource types*. This choice allowed us to explore the design of a purely functional standalone language, in an area where it is more common to embed a functional core in an imperative shell. The language borrows the combinators of arrowized FRP, but has discrete-time semantics. The design of the full set of combinators is work in progress, driven by examples. So far, we have evaluated Hailstorm by writing standard examples from the literature (earthquake detection, a railway crossing system and various other clocked systems), and also running examples on the GRiSP embedded systems board, through generation of Erlang.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Domain specific languages**; • **Computer systems organization → Sensors and actuators**; *Embedded software.*

KEYWORDS

functional programming, IoT, compilers, embedded systems

ACM Reference Format:
Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3414080.3414092

PPDP '20, September 8–10, 2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09…$15.00
https://doi.org/10.1145/3414080.3414092

1 INTRODUCTION

As the density of IoT devices and diversity in IoT applications continue to increase, both industry and academia are moving towards decentralized system architectures like *edge computing* [38]. In edge computation, devices such as sensors and client applications are provided greater computational power, rather than pushing the data to a backend cloud service for computation. This results in improved response time and saves network bandwidth and energy consumption [50]. In a growing number of applications such as aeronautics and automated vehicles, the real-time computation is more robust and responsive if the edge devices are compute capable.

In a more traditional centralized architecture, the sensors and actuators have little logic in them; they rather act as data relaying services. In such cases, the firmware on the devices is relatively simple and programmed almost exclusively using the C programming language. However with the growing popularity of edge computation, more complex logic is moving to the edge IoT devices. In such circumstances, programs written using C tend to be verbose, error-prone and unsafe [17, 27]. Additionally, IoT applications written in low-level languages are highly prone to security vulnerabilities [7, 58].

Hailstorm is a domain-specific language that attempts to address these issues by bringing ideas and abstractions from the functional and reactive programming communities to programming IoT applications. Hailstorm is a *pure, statically-typed* functional programming language. Unlike *impure* functional languages like ML and Scheme, Hailstorm restricts arbitrary side-effects and makes dataflow explicit. The purity and static typing features of the language, aside from providing a preliminary static-analysis tool, provide an essential foundation for embedding advanced language-based security techniques [54] in the future.

The programming model of Hailstorm draws inspiration from the extensive work on Functional Reactive Programming (FRP) [18]. FRP provides an interface to write *reactive* programs such as graphic animations using (1) continuous time-varying values called *Behaviours* and (2) discrete values called *Events*. The original formulation of FRP suffered from a number of shortcomings such as space-leaks [41] and a restrictive form of *stream based I/O*.

A later FRP formulation, *arrowized* FRP [46], fixed space leaks, and in more recent work Winograd-Cort et al. introduced a notion of *resource types* [66] to overcome the shortcomings of the stream based I/O model. The work on resource types is a library in Haskell, and is not suitable to run directly on resource-constrained hardware. Hailstorm uses resource types to uniquely identify each I/O

resource. It treats each resource as a signal function to track its lifetime, and prevents resource duplication through the type system. Hailstorm currently has a simple discrete time semantics, though we hope to explore extensions later.

A Hailstorm program is compiled to a dataflow graph, which is executed synchronously. The core of the language is a pure call-by-value implementation of the lambda calculus. The synchronous language of arrowized-FRP provides a minimal set of combinators to which the pure core constructs of Hailstorm can be raised. This language of arrows then enforces a purely functional way to interact with I/O, using resource types.

Hailstorm, in its current version, is a work-in-progress compiler which does not address the reliability concerns associated with node and communication failures plaguing edge devices [58]. We discuss the future extensions of the language to tackle both reliability and security concerns in Section 7. We summarize the contributions of Hailstorm as follows:

- **A statically-typed purely functional language for IoT applications.** Hailstorm provides a tailored, purely functional alternative to the current state of programming resource constrained IoT devices.
- **Resource Types based I/O.** Hailstorm builds on Winograd-Cort et al's work to provide the semantics and implementation of an alternate model of I/O for pure functional languages using *resource types* - which fits the streaming programming model of IoT applications. (Section 4.1)
- **Discrete time implementation** Hailstorm uses the combinators of arrowized FRP in a discrete time setting (Section 3).
- **An implementation of the Hailstorm language.** We implement Hailstorm as a standalone compiler, with Erlang and LLVM backends. We have run case studies on the GRiSP embedded system boards [60], to evaluate the features of the language. (Section 4). The compiler implementation and the examples presented in the paper are made publicly available[1].

2 LANGUAGE OVERVIEW

In this section we demonstrate the core features and syntax of Hailstorm using running examples. We start with a simple pure function that computes the n^{th} Fibonacci number.

```
def main : Int = fib 6

def fib : (Int -> Int)
  = fib_help 0 1

def fib_help (a : Int) (b : Int) (n : Int) : Int
  = if n == 0
    then a
    else fib_help (a + b) a (n - 1)
```

The simple program above, besides showing the ML-like syntax of Hailstorm, demonstrates some features like (1) higher-order functions (2) recursion (3) partial application (4) tail-call optimization and (5) static typing. All top-level functions in a Hailstorm program have to be annotated with the types of the arguments and the return type, which currently allows only monomorphic types. However certain built-in combinators supported by the language are polymorphic which will be discussed in the following section.

The pure core of the language only allows writing pure functions which have no form of interactions with the outside world. To introduce I/O and other side effects, we need to describe the concept of a *signal function*.

2.1 Signal Functions

A fundamental concept underlying the programming model of Hailstorm is that of a *Signal Function*. Signal Functions, derived from the work on arrowized-FRP [46], are functions that *always* accepts an input and *always* returns an output.

Signal functions are analogous to the nodes of a dataflow graph. Signal functions operate on *signals* which do not have any concrete representation in the language. A signal denotes a discrete value at a give point of time. Nilsson et al [46] use the electric circuit analogy: a signal corresponds to a wire and the current flowing through it, while signal functions correspond to circuit components. An important distinction between Hailstorm and both classic and arrowized-FRP is that signals are always treated as discrete entities in Hailstorm unlike the continuous semantics enforced by FRP.

To create larger programs Hailstorm provides a number of built-in combinators to compose signal functions. These combinators are drawn from the Arrow framework [31] which is a generalization of monads. Arrows allow structuring programs in a *point-free* style, while providing mathematical laws for composition. We start by presenting some of the core Hailstorm combinators[2] and their types for composing signal functions.

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

Some of the built-in combinators in Hailstorm are polymorphic and the type parameters `a, b` and `c` represent the polymorphic types. `mapSignal#` is the core combinator which lifts a pure Hailstorm function to the synchronous language of arrows, as a signal function (See Fig 1).

Hailstorm then provides the rest of the built-in combinators to compose signal functions while satisfying nine arrow laws [39]. One of the advantages of having a pure functional language is that such laws can be freely used by an optimizing compiler to aggressively inline and produce optimized code. The semantics of composing signals with the arrow combinators is visually depicted in Fig 1.

[1] https://abhiroop.github.io/ppdp-2020-artifact.zip

[2] Non-symbolic built-in combinators & driver functions in Hailstorm end with #

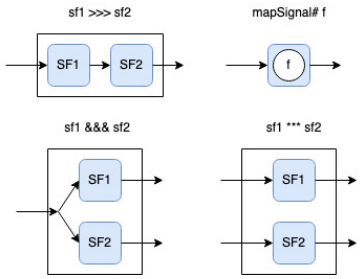


Figure 1: Arrow combinators for signal functions

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

2.2 I/O

Hailstorm adopts a streaming programming model, where an effectful program is constructed by composing various signal functions in the program. The final program is embedded in a stream of input flowing in and the program transforms that into the output.

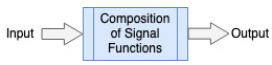


Figure 2: A Hailstorm program interacting with the real world

While this model of I/O adapts well with a pure functional language, it is reminiscent of the now abandoned stream-based I/O interface in Haskell. In Haskell's early stream-based I/O model, the type of the `main` function was `[Response] → [Request]`. The major problems with this model are :

- All forms of I/O are restricted to happen at the main function leading to non-modular programs, especially in case of applications running on IoT devices where I/O functions dominate the majority of the program.
- It is non-extensible as the concrete types of `Request` and `Response` need to be altered every time any new I/O facility has to be added [48].
- There is also no clear mapping between an individual `Request` and its `Response` [48].

Work by Winograd-Cort et al. on resource types [66] attempts to address this problem by *virtualizing* real-world devices as signal functions. What we mean here by "virtualizing" is that the scope of a program is extended so that devices like sensors and actuators are represented using signal functions. For example:

```
sensor : SF () Int
uart_rx  : SF () Byte
actuator : SF Bool ()
```

We adopt this model in Hailstorm. The type parameter `()` represents a *void* type. There are no *values* in the language which inhabit the `()` type. The `()` type always appears within a signal function. So, an example like `sensor : SF () Int` - this represents an *action* which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a *value* and an *action*. When an action (like `sensor : SF () Int`) is evaluated, it returns a representation of that function call rather than actually *executing* the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

```
-- Expression 1 : accepts an input and
-- duplicates that input to return a pair
let x = getInput -- makes one I/O call
 in (x,x)

-- Expression 2 : accepts two inputs
-- returns both of them as a pair
(getInput, getInput) -- makes two I/O calls
```

After enforcing a difference between values and action in the language, we soon encounter one of the pitfalls of treating a real-world object as a virtual device - it allows a programmer to write programs with unclear semantics. For example:

```
def foo : SF () (Int, Int) = sensor &&& sensor
```

Although the above program is currently type correct, it can have two conflicting semantics - (1) either `sensor &&& sensor` implies two consecutive calls to the sensor device or (2) a single call emitting a pair. Given the type of the `sensor` function, the latter is not supported and the former is incompatible with Hailstorm's discrete, synchronous semantics.

The notion of *Resource Types* seeks to solve this problem by labeling each device with a type-level identifier, such that duplicating a device becomes impossible in the program. We change the type of `sensor` to :

```
resource S

sensor : SF S () Int
```

The *resource* keyword in Hailstorm declares a type level identifier which is used for labeling signal functions like `sensor` above. All the built-in arrow combinators introduced previously are now enriched with new *type-level* rules for composition as follows:

```
mapSignal# : (a -> b) -> SF Empty a b
(>>>) : SF r1 a b -> SF r2 b c -> SF (r1 ∪ r2) a c
(&&&) : SF r1 a b -> SF r2 a c -> SF (r1 ∪ r2) a (b, c)
(***) : SF r1 a b -> SF r2 c d -> SF (r1 ∪ r2) (a, c) (b, d)
```

In the combinators above, the type parameters r_1, r_2 represent polymorphic resource type variables, which act as labels for the effectful signal functions. The combinator `mapSignal#` lifts a pure

Hailstorm function to a signal function without any effectful operations, and as a result the resource type is `Empty`. The combinators `>>>`, `&&&` and `***` compose signal functions, and result in a disjoint-union of the two resources types. This type-level disjoint union prevents us from copying the same resource using any of these combinators. So in Hailstorm if we try this,

```
def foo : SF (S ∪ S) () (Int, Int) = sensor &&& sensor
```

we currently get the following upon compilation:

```
Type-Checking Error:
  Error in "foo":
   Cannot compose resources : S S containing same resource
   Encountered in
      sensor &&& sensor
```

The type rules associated with composing Hailstorm combinators and their operational semantics are presented formally in Section 3.2 and 3.3 respectively.

2.2.1 Example of performing I/O. We can distinguish the read and write interface of a resource using two separate resource types. For example, to repeatedly blink an LED we need two APIs - (1) to *read* its status (2) to *write* to it. The drivers for these two functions have the following types:

```
readLed# : SF R () Int
writeLed# : SF W Int ()
```

We use the integer 1 to represent light ON status and 0 for OFF. The program for blinking the LED would be:

```
def main : SF (R ∪ W) () () =
  readLed# >>> mapSignal# flip >>> writedLed#

def flip (s : Int) : Int =
  if (s == 0) then 1 else 0
```

The above program runs the function `main` infinitely. It is possible to adjust the rate at which we want to run this program, discussed later in Section 2.5. This treatment of I/O as signal functions has the limitation that each device (as well as their various APIs) has to be statically encoded as a resource type in the program.

2.3 State

Hailstorm supports stateful operations on signals using the `loop#` combinator.

```
loop# : c -> SF Empty (a, c) (b, c) -> SF Empty a b
```

The type of the `loop#` combinator is slightly different from the type provided by the `ArrowLoop` typeclass in Haskell, in that it allows initializing the state type variable `c`. The internal body of the signal function encapsulates a polymorphic state entity. This entity is repeatedly fed back as an additional input, upon completion of a whole step of signal processing by the entire dataflow graph. Fig 3 represents the combinator visually.

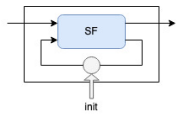


Figure 3: The stateful loop# function

The `loop#` combinator can be used to construct the `delay` function as found in synchronous languages like Lustre [26], for encoding state.

```
def delay (x : Int) : SF Empty Int Int =
  loop# x (mapSignal# swap)

def swap (a : Int, s : Int) : (Int, Int) = (s, a)
```

2.4 A sample application

We now demonstrate the use of the Hailstorm combinators in a sample application. The application that we choose is a simplified version of an earthquake detection algorithm [65] which was first used by Mainland et al. to demonstrate their domain specific language for wireless sensor networks [42]. The figure below shows the core dataflow graph of the algorithm.

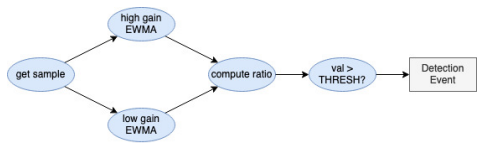


Figure 4: The earthquake detection dataflow graph

The exponentially weighted moving average (EWMA) component above is a stateful element. We assume the `getSample` input function is a wrapper around a seismometer providing readings of discrete samples. At the rightmost end, the `Detection Event` would be another stateful entity which would include some form of an *edge detector*. The entire program for the earthquake detection is given in Fig 5.

The function edge in Fig 5 is a stateful edge detector which generates an action if a boolean signal changes from `False` to `True`. To program this, we use an imaginary actuator (like an LED) in a GRiSP board, which would glow red once if the input to it is 1, signalling danger, and otherwise stay green signalling no earthquake.

2.5 Sampling rate

The combinators introduced so far execute *instantaneously* using a logical clock. In Hailstorm, one logical time step includes the following actions, in sequence -

- accepting a discrete sample of data from each of its connected input devices
- passing the discrete sample through the dataflow graph
- finally passing a discrete value to the responsible actuator

```
resource S
resource E

def main : SF (S ∪ E) () () =
 getSample >>> detect >>> edge

def detect : SF Empty Float Bool
  =   (ewma high &&& ewma low)
  >>> (mapSignal# (\(hi : Float, lo : Float) =>
                    (hi / lo) > thresh))

def ewma (α : Float) : SF Empty Float Float
  = let func = \(x : Float, x_old : Float) =>
                let x_new = (α *. x) +. (1.0 -. α) *. x_old
                 in (x_new, x_new)
    in loop# 0.0 (mapSignal# func)

-- constants
def low : Float = ...
def high : Float = ...
def thresh : Float = ...

def edge : SF E Bool () =
  loop# False (mapSignal# edgeDetector) >>>
  actuator

def edgeDetector (a : Bool, c : Bool) : (Int, Bool) =
  if (c == False && a == True)
  then (1, a)
  else (0, a)

-- getSample : SF S () Float - Erlang driver
-- actuator : SF E Int () - Erlang driver
```

Figure 5: The earthquake detection algorithm

A program returning a signal function continuously loops around, streaming in input and executing the above steps, at the speed it takes for the instructions to execute. However under most circumstances we might wish to set a slower rate for the program. The `rate#` combinator is used for that purpose,

```
rate# : Float -> SF r a b -> SF r a b
```

The first argument to `rate#` is the length of the wall clock time (in seconds) at which we wish to set the period of sampling input. This helps us establish a relation between the wall clock time and Hailstorm's logical clock. We demonstrate the utility of the *rate#* combinator using a *Stopwatch* example.

2.5.1 Stopwatch. We program a hypothetical stopwatch which accepts an input stream of `Int`s where 1 represents START, 2 represents RESET and 3 represents STOP.

```
def f (g : Float) (a : Int, c : Float) : (Float, Float) =
  let inc = c +. g in
  case a of
       1 ~> (inc, inc);
       2 ~> (0.0,0.0);
       _ ~> (c,c)

def stopwatch (g : Float) : SF Empty Int Float
  = rate# g (loop# 0.0 (mapSignal# (f g)))

def main : SF (I U O) () () =
  input >>> stopwatch 1.0 >>> output
```

In the above program, the `rate#` combinator uses the argument `g` to set the sampling rate to 1 second, which in turn fixes the granularity of the stopwatch as 1 second.

2.5.2 Limitation. In the current implementation of Hailstorm, an operation like `(rate# t2 (rate# t1 sf1))` would result in setting the final sampling rate as t_2, overwriting the value of t_1. In the programs presented here, we use a single clock, and hence a single sampling rate. Libraries like Yampa [14] provide combinators like *delay* :: $Time \rightarrow a \rightarrow SF\ a\ a$ which allow *oversampling* for dealing with multiple discrete sampling rates. As future work, we hope to adopt oversampling operators for communication among signal functions with different sampling rates.

2.6 Switches

A Hailstorm dataflow graph allows a form of *dynamic*, data-driven switching within the graph. It accomplishes this using the `switch#` combinator:

```
switch# : SF r1 a b -> (b -> SF r2 b c) -> SF (r1 ∪ r2) a c
```

The first argument to `switch#` accepts a signal function whose output data is used to switch between the various signal functions. The strict typing of Hailstorm restricts the branches of the switch to be of the same type, including the resource type. The switching dataflow is visually presented in Fig 6.

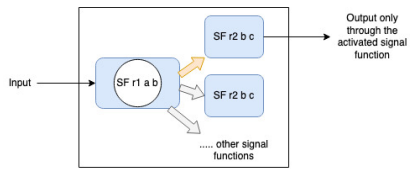


Figure 6: A switch activating only the top signal function

2.6.1 Limitation. `switch#` is a restrictive combinator with a number of known limitations:
- `switch#` constrains all of its branches to be of the same type. This is particularly restrictive when dealing with actuators where each actuator would have their own resource type. We currently deal with the notion of *choice* in the following way -

```
... >>> foo >>> (actuator1 *** actuator2)
-- instead of emitting a single value foo will emit a pair
-- of values encoded as (move actuator1, dont move actuator2)
```

The Arrow framework provides more useful combinators based on the `ArrowChoice` typeclass which are currently absent from Hailstorm.

- The `switch#` combinator is an experiment to describe expressions of the form:

```
switch# input_signal_function
  (\val => if <condition_1 on val>
           then SF_1
           else if <condition_2 on val>
                then SF_2
                else  ...)
```

The combinator currently supports expression only of the above form. Thus, we currently do not allow more general functions of type (`b→SF b c`) as the second parameter to `switch#`, and so avoid problems with possibly undefined runtime behaviour. However, there is no check in the compiler to enforce this restriction. As future work we hope to adopt the *guards* syntax of Haskell to represent the second parameter as a collection of boolean clauses and their corresponding actions.

3 SYNTAX, SEMANTICS AND TYPES

In the previous sections, we have given an informal treatment to the most important syntactic parts of Hailstorm, for IoT applications, and described the programming model using them. In this section, we present the core syntax, type rules and operational semantics of the main parts of Hailstorm.

3.1 Syntax

The set of types in the source language is given by the following grammar.

$$\tau ::= () \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid \tau_1 \rightarrow \tau_2 | \tau_1 \overset{r}{\Longrightarrow} \tau_2 | (\tau_1, \tau_2)$$

The type $\tau_1 \overset{r}{\Longrightarrow} \tau_2$ represents a signal function type from `a` to `b` with the resource type `r` i.e `SF r a b`.

The abstract syntax of the core *expressions* of Hailstorm is given by the following grammar. The meta-variable $x \in Var$ ranges over variables of the source language. Additionally we let $i \in \mathbb{Z}$ and $f \in \mathbb{R}$. We use e and e_{sf} separately to denote ordinary expressions and arrow based signal function expressions respectively.

$$\begin{aligned}
e &::= x|\bar{i}|\bar{f}|\texttt{True}|\texttt{False}|i_1\ binop_i\ i_2|f_1\ binop_f\ f_2 \\
&\quad |e_1\ relop\ e_2|\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ |\lambda x : \tau.\ e\ |\ (e_1\ e_2) \\
&\quad |\ \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2|\ (e_1,\ e_2)\ |\ \texttt{fst\#}\ e\ |\ \texttt{snd\#}\ e \\
e_{sf} &::= \texttt{mapSignal\#}\ (\lambda x.e)|e_{sf1} >>> e_{sf2} \\
&\quad |e_{sf1}\ \&\&\&\ e_{sf2}|e_{sf1} * * * e_{sf2}|\texttt{loop\#}\ e_1\ e_2 \\
&\quad |\texttt{switch\#}\ e_1\ (\lambda x.e)|\texttt{read\#}|\texttt{write\#} \\
binop_i &::= + \mid - \mid * \\
binop_f &::= +. \mid -. \mid *. \mid / \\
relop &::= > \mid < \mid >= \mid =< \mid ==
\end{aligned}$$

In the grammar above we describe two primitives for I/O called `read#` and `write#`. In practise, as Hailstorm deals with a number of I/O drivers there exists a variety of I/O primitives with varied parameters and return types. However for the purpose of presenting the operational semantics, we abstract away the complexity of the drivers and use the abstracted `read#` and `write#` to describe the semantics in Section 3.3.

3.2 Type rules

Hailstorm uses a fairly standard set of type rules except for the notion of *resource types*. The typing context of Hailstorm employs *dual contexts*, in that it maintains (1) Γ - a finite map from variables to their types and (2) Δ - a finite set which tracks all the I/O resources connected to the program.

$$\Delta; \Gamma ::= \cdot \mid \Gamma, x : \tau$$

An empty context is given by $\cdot$. Additionally $dom(\Gamma)$ provides the set of variables bound by a typing context.

In Fig 7, we show the most relevant type rules concerning signal functions and their composition. The remaining expressions follow standard set of type rules which is provided in its entirety in the extended version of this paper [55].

Looking at the rule `T-Mapsignal`, a signal function such as $\tau_1 \overset{\emptyset}{\Longrightarrow} \tau_2$ denotes the result of applying `mapSignal#` to a pure function. This results in an expression with an empty resource type denoted by $\emptyset$. A missing rule is the introduction of a new resource type in the resource type context Δ. The resource type context is an append-only store and a new resource is introduced using the keyword `resource`. It can be defined using this simple reduction semantics

$$\frac{}{\Delta; \Gamma \vdash resource\ r\ \rightsquigarrow\ \Delta \cup r; \Gamma}$$

where $\rightsquigarrow$ denotes one step of reduction which occurs at compile-time. The rules such as `T-Compose, T-Fanout, T-Combine, T-Switch` apply a type-level *disjoint union* to prevent resource duplication.

3.3 Big-step Operational Semantics

In this section, we provide a big-step operational semantics of our implementation of the Hailstorm language, by mapping the meaning of the terms to the lambda calculus. We begin by defining the *values* in lambda calculus that cannot be further reduced:

$$V ::= i \mid f \mid \lambda x.E$$

i and f represent integer and float constants respectively. We use n to represent variables and the last term denotes a lambda expression. The syntax that we use for defining our judgements is of the form :

$$s_1 \vdash M \Downarrow V_i, s_2$$

The variables s_1, s_2 are finite partial functions from variables `n` to their bound values $V_i \in V$. In case a variable n is unbound and s is called with that argument it returns $\emptyset$. The above judgement is read as *starting at state s_1 and evaluating the term M results in the irreducible value $V_i \in V$ while setting the final state to s_2.*

The first judgement essential for our semantics is this,

$$\frac{}{s \vdash V \Downarrow V, s}$$

which means that the values V cannot be reduced further.

We use a shorthand notation $\rho(n, s)$ to signify *lookup the variable n in s.* Additionally, to make the semantics more compact we

$$\frac{\Delta;\Gamma \vdash e : \tau_1 \to \tau_2}{\Delta;\Gamma \vdash mapSignal\#\ e : \tau_1 \overset{\emptyset}{\Longrightarrow} \tau_2}\ \text{(T-Mapsignal)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_2 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ >>>\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} \tau_3}\ \text{(T-Compose)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_1 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ \&\&\&\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} (\tau_2, \tau_3)}\ \text{(T-Fanout)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_3 \overset{r_2}{\Longrightarrow} \tau_4 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ ***\ e_2 : (\tau_1, \tau_3) \overset{r_1 \cup r_2}{\Longrightarrow} (\tau_2, \tau_4)}\ \text{(T-Combine)}$$

$$\frac{\Delta;\Gamma \vdash e : (\tau_1,\ \tau_c) \overset{\emptyset}{\Longrightarrow} (\tau_2,\ \tau_c) \qquad \Delta;\Gamma \vdash c : \tau_c}{\Delta;\Gamma \vdash loop\#\ c\ e : \tau_1 \overset{\emptyset}{\Longrightarrow} \tau_2}\ \text{(T-Loop)} \qquad \frac{\Delta;\Gamma \vdash t : Float \qquad \Delta;\Gamma \vdash e : \tau_1 \overset{r}{\Longrightarrow} \tau_2 \qquad r \in \Delta}{\Delta;\Gamma \vdash rate\#\ t\ e : \tau_1 \overset{r}{\Longrightarrow} \tau_2}\ \text{(T-Rate)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_2 \to \tau_2 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash switch\#\ e_1\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} \tau_3}\ \text{(T-Switch)}$$

$$\frac{r \in \Delta}{\Delta;\Gamma \vdash read\# : () \overset{r}{\Longrightarrow} \tau}\ \text{(T-Read)} \qquad \frac{r \in \Delta}{\Delta;\Gamma \vdash write\# : \tau \overset{r}{\Longrightarrow} ()}\ \text{(T-Write)}$$

Figure 7: Typing rules of signal functions in Hailstorm

$$\frac{s \vdash exp \Downarrow \lambda x.E, s}{s \vdash mapSignal\#\ exp \Downarrow \lambda x.E, s}\ \text{(eval-Mapsignal)} \qquad \frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ >>>\ exp_2 \Downarrow \lambda x.V_2\ (V_1\ x), s_3}\ \text{(eval-Compose)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ \&\&\&\ exp_2 \Downarrow \lambda x.\ <V_1\ x, V_2\ x>, s_3}\ \text{(eval-Fanout)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ ***\ exp_2 \Downarrow \lambda x.\lambda y.\ <V_1\ x, V_2\ y>, s_3}\ \text{(eval-Combine)}$$

$$\frac{s_1 \vdash init \Downarrow V_i, s_2 \qquad s_2 \vdash exp \Downarrow V_1, s_3 \qquad n \notin dom(s_3)}{s_1 \vdash loop\#_n\ init\ exp \Downarrow \lambda x.fst\ (V_1\ (x,\ V_s)), s_3[n \mapsto snd\ (V_1\ (x,\ V_i))]}\ \text{(eval-Loop-Init)}$$

$$\frac{s_1 \vdash init \Downarrow V_i, s_2 \qquad s_2 \vdash exp \Downarrow V_1, s_3 \qquad n \in dom(s_3)}{s_1 \vdash loop\#_n\ init\ exp \Downarrow \lambda x.fst\ (V_1\ (x,\ \rho(n, s_3))), s_3[n \mapsto snd\ (V_1\ (x,\ \rho(n, s_3)))]}\ \text{(eval-Loop)}$$

$$\frac{s_1 \vdash e_2 \Downarrow V, s_2 \qquad s_2 \vdash e_1 \Downarrow \lambda x.E_1, s_3 \qquad s_3 \vdash E_1[x \mapsto V] \Downarrow V_f, s_4}{s_1 \vdash (e_1\ e_2) \Downarrow V_f, s_3}\ \text{(eval-App)} \qquad \frac{s_1 \vdash t \Downarrow V_t, s_2[\Psi \mapsto V_t] \qquad s_2[\Psi \mapsto V_t] \vdash exp \Downarrow V, s_3}{s_1 \vdash rate\#\ t\ exp \Downarrow V, s_3}\ \text{(eval-Rate)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow \lambda b.\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n], s_3}{s_1 \vdash switch\#\ exp_1\ exp_2 \Downarrow \lambda a.((\lambda b.\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n])(V_1\ a))\ (V_1\ a), s_3}\ \text{(eval-Switch)}$$

$$\frac{}{s \vdash read\# \Downarrow \lambda x.read, s}\ \text{(eval-Read)} \qquad \frac{}{s \vdash write\# \Downarrow \lambda x.(write\ x), s}\ \text{(eval-Write)}$$

Figure 8: Big-Step Operational Semantics of signal functions in Hailstorm

use pairs `<a,b>` and their first and second projections, `fst, snd`. They do not belong to `V` but it is possible to represent all three of them using plain lambdas and function application - shown in the extended version of this paper [55].

In Fig 8, we show the most relevant big-step operational semantics concerning signal function based combinators. The remaining expressions have standard semantics and the complete rule set is provided in the extended paper [55]. In the rule `eval-Rate`, we use Ψ to store the sampling rate. In our current implementation, when composing signal functions with different sampling rates, the state transition from s_2 to s_3 overwrites the first sampling rate.

In `eval-Loop-Init` and `eval-Loop`, the subscript `n` represents a variable name that is used as a key, in the global state map `s`, to identify each individual state.

For the rule `eval-Switch`, $\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n]$ represents a conditional expression that uses the value of `b` i.e. $(V_1\ a)$ to *choose* one of the several branches - $\lambda c.E_i$ - and then supplies $(V_1\ a)$ again to the selected branch to actually generate a value of the stream.

Of special interest in Fig 8 are the rules `eval-Read` and `eval-Write`. We need to extend our lambda calculus based abstract machine with the operations, `read` and `write`, to allow any form of I/O. The effectful operations, `read` and `write`, are guarded by λs to prevent

any further evaluation, and as a result are treated as values. This method is essential to ensure the purity of the language - by treating effectful operations as values.

The program undergoes a *partial evaluation* transformation which evaluates the entire program to get rid of all the λs guarding the `read` operations. Given the expression $\lambda x.read$ the compiler supplies a compile time token of type `()` which removes the *lambda* and exposes the effectful function `read`. The partially evaluated program is then prepared to conduct I/O. This approach is detailed further in Section 4.1.

The big-step semantics of the language shows its evaluation strategy. However to understand the streaming, infinite nature of an effectful Hailstorm program we need an additional semantic rule. A Hailstorm `function` definition is itself an expression and a `program` is made of a list of such functions,

$$Program ::= main : [Function]$$
$$Function ::= e | e_{sf}$$

Each Hailstorm program compulsorily has a `main` function. After the entire program is *partially evaluated* (described in Section 4.1) and given that the `main` function causes a side effect (denoted by `()` below), we have the following rule:

$$\frac{s_1 \vdash main \Downarrow (), s_2}{s_1 \vdash main \Downarrow main, s_2} \text{ (eval-Main)}$$

The `eval-Main` rule demonstrates the streaming and infinite nature of a Hailstorm program when the main function is a signal function itself. After causing a side effect, it calls itself again and continues the stream of effects while evaluating the program using the semantics of Fig 8.

4 IMPLEMENTATION

Here we describe an implementation of the Hailstorm language and programming model presented in the previous sections. We implement the language as a *compiler* - the Hailstorm compiler - whose compilation architecture is described below.

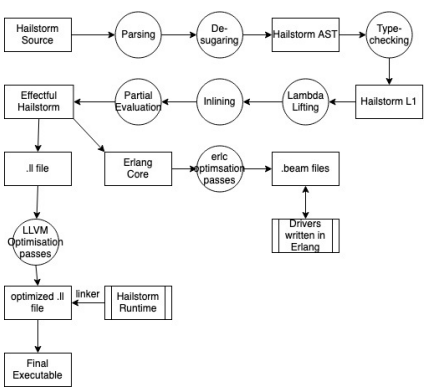


Figure 9: The Hailstorm compilation architecture

The compiler pipeline starts by parsing a Hailstorm source file and *desugaring* small syntactic conveniences provided to reduce code size (such as `case` expressions). After desugaring, the Hailstorm AST constituted of the grammar described in Section 3.1 is generated. Next, typechecking of the AST using the type rules of Fig 7 is done. After a program is typechecked, it is transformed into the Hailstorm core language called `L1`.

The `L1` language is an enriched version of the simply typed lambda calculus (STLC) with only nine constructors. Unlike STLC, `L1` supports recursion by calling the name of a global function. It is currently incapable of recursion using a `let` binding. `L1` has a simpler type system than the Hailstorm source, as it *erases* the notion of resource types - which are exclusively used during type checking.

The `L1` language being considerably simpler, forms a sufficient foundation for running correctness preserving optimization passes. `L1` attempts to get rid of all closures with free variables, as they are primarily responsible for dynamic memory allocation. Additionally, it attempts to inline expressions (primarily partial applications) to further reduce both stack and heap memory allocation. The optimization passes are described in further detail in Section 4.2.

The final optimization pass in `L1` is a *partial evaluation* pass which specializes the program to convert it into an effectful program. Before this pass, all I/O inducing functions are treated as values, by guarding them inside a λ abstraction. This pass evaluates those expressions by passing a compile-time token and turning the `L1` non-effectful program into an effectful one. This pass is discussed in further detail in Section 4.1.

Finally, the effectful Hailstorm program gets connected to one of its backends. Recursion is individually handled in each of the backends by using a global symbol table. We currently support an LLVM backend [36] and a BEAM backend [4].

4.1 I/O

The I/O handling mechanism of the Hailstorm compiler is the essential component in making the language *pure*. A pure functional language allows a programmer to equationally reason about their code. The entire program is written as an *order independent* set of equations. However, to perform I/O in a programming language, it is necessary to (1) enforce an order on the I/O interactions, as they involve *chronological* effects visible to the user and (2) interact with the real world and actually perform an effect.

To solve (1) we use the `eval-App` rule given in Fig 8. Hailstorm, being a *call-by-value* implementation of the lambda calculus, follows *ordering* in function application by always evaluating the function argument before passing it to the function. This semantics of function application allows us to introduce some form of ordering to the equations.

To solve (2) we have extended our pure lambda calculus core to involve effectful operations like `read` and `write`. Again, as observed in the semantic rules, `eval-Read` and `eval-Write` from Fig 8, the effectful operations are guarded by λ abstractions to treat them as values, rather than operations causing side-effects. This allows us to freely inline or apply any other optimization passes while preserving the correctness of the code. However, to finally perform the side effect, we need to resolve this lambda abstraction at compile

time. We do this by *partially evaluating* [33] our program. Below we describe the semantics of the partial evaluation step.

A Hailstorm program is always enforced (by the typechecker in our implementation) to contain a *main* function. We shall address the case of an effectful program i.e a *main* function whose return type is an effectful signal function such as `SF r () ()`. The `()` type in the input and output parameters reflect that this function reads from a real world source like a sensor and causes an effect such as moving an actuator. The partial application pass is only fired when the program contains the `()` type in either one of its signal function parameter.

Let us name the main function above with the return type of `SF r () ()` as $main_{sf}$. Now $main_{sf}$ is itself a function, embedded in a stream of input and transforming the input to an output, causing an effect. As it is a function we can write: $main_{sf} = \lambda t.E$.

Additionally in our implementation, after typechecking, a signal function type $\tau_1 \overset{r}{\Longrightarrow} \tau_2$ is reduced to a plain arrow type - $\tau_1 \to \tau_2$. Now, using the two aforementioned definitions, we can write the following reduction semantics:

$$\frac{main_{sf} = \lambda t.E \qquad main_{sf} : () \overset{r}{\Longrightarrow} () \qquad \tau_1 \overset{r}{\Longrightarrow} \tau_2 \rightsquigarrow \tau_1 \to \tau_2}{\lambda t.E : () \overset{r}{\Longrightarrow} () \rightsquigarrow \lambda t.E : () \to ()} \quad \text{(eval-Partial)}$$
$$\frac{}{\lambda t.E : () \to () \rightsquigarrow E[t \mapsto \theta : ()]}$$

where $\rightsquigarrow$ denotes one step of reduction and θ denotes an arbitrary compile time token of type `()`. The final step, which produces the expression $E[t \mapsto \theta : ()]$, is the partial evaluation step. We demonstrate this reduction semantics in action using an example:

```
read#  : SF STDIN () Int
write# : SF STDOUT Int ()

def main : SF (STDIN U STDOUT) () () = read# >>> write#
```

We use the `eval-Read`, `eval-Write` and `eval-Compose` rules to translate the *main* function above to

```
main = λt. (λy. write y) ((λx. read) t)
```

Given the above definition of *main*, we can apply the reduction rule `eval-Partial` and `eval-App` to get the following,

$$\frac{}{\lambda t.\ (\lambda y.\ write\ y)\ ((\lambda x.\ read)\ t) \rightsquigarrow_* (\lambda y.\ write\ y)\ (read)} \quad \text{(Partial evaluation)}$$

Now the program is ready to create a *side effect* as the `read` function is no longer guarded by a λ abstraction. The `eval-App` rule guarantees that `read` is evaluated first and only then the value is fed to `write` owing to the *call-by-value* semantics of Hailstorm.

4.1.1 Limitation. The Hailstorm type system doesn't prevent a programmer from writing `write# >>> read#`. The type of such a program would be `SF (STDOUT U STDIN) Int Int`. As the types do not reflect the `()` type, the partial evaluation pass is not fired and the program simply generates an unevaluated closure - which is an expected result given the meaningless nature of the program. However, the type would allow composing it with other pure functions and producing bad behaviour if those programs are connected to meaningful I/O functions. This is simply solved by adding the following type rule:

$$\frac{r \in \Delta \qquad r \neq \emptyset \qquad \Delta;\Gamma \vdash \tau_1 = () \vee \tau_2 = ()}{exp : \tau_1 \overset{r}{\Longrightarrow} \tau_2} \quad \text{(T-Unsafe)}$$

4.2 Optimizations

4.2.1 Lambda Lifting. Hailstorm, being a functional language, supports higher order functions (HOFs). HOFs frequently capture free variables which survive the scope of a function call. For example:

```
1 def addFive (nr : Int) : Int =
2   let x = 5 in
3   let addX = \(y:Int) => x + y in
4   addX nr
```

Our implementation treats HOFs as closures which are capable of capturing an environment by allocating the environment on the heap. In line no. 3 above, the value of the variable `x` is heap allocated. However, in resource constrained devices heap memory allocation, is highly restrictive and any language targeting such devices should attempt to minimize allocation.

Hailstorm applies a *lambda-lifting* [32] transformation to address this. Lambda-lifting lifts a lambda expression with free variables to a top-level function and then updates related call sites with a call to the top-level function. The free variables then act as arguments to the function. This effectively allocates them on the stack (or registers). Owing to our restricted language and the lack of polymorphism, our algorithm is less sophisticated than the original algorithm devised by Johnsson. We describe the operational semantics of our algorithm in the `lambda-lifting` rule below.

We use a slightly different notation from Section 3.3 here. P_n is used to describe the entire program with its collection of top level functions. We identify a modified program with $P_n[G(x).E]$ to mean a new program with an additional global function `G` which accepts an argument `x` and returns an expression `E`. Finally the fv function is used find the set of free variables in a Hailstorm expression.

$$\frac{P_1 \vdash exp \Downarrow \lambda x.\ E,\ P_2 \qquad fv(\lambda x.\ E) \neq \emptyset \qquad fv(\lambda x.\ E) = \{i_1, ..., i_n\} \qquad P_2 \vdash \lambda x.\ E \Downarrow E',\ P_3[G(i_1, ..., i_n, x)\ .\ E]}{P_1 \vdash exp \Downarrow E',\ P_3[G(i_1, ..., i_n, x)\ .\ E]}$$

The above rule returns a modified expression E' which consists of a function call to $G(i_1, ..., i_n, x)\ .\ E$ with the free variables $i_1, ..., i_n$ as arguments. This rule is repeatedly run on local lambda expressions until none has any free variables. It transforms the program `foo` above to :

```
def addX' (x:Int) (y:Int) = x + y

def addFive (nr : Int) : Int =
  let x = 5 in
  addX' x nr
```

4.2.2 Inlining. The inlining transformation works in tandem with the lambda lifter to reduce memory allocations. Inlining a lambda calculus based language reduces to plain β reduction. i.e $((\lambda x.\ M)\ E) \rightsquigarrow M[x \mapsto E]$. Inlining subsumes optimization passes like *copy-propagation* in a functional language. Our prototype inliner is relatively conservative in that (1) it doesn't attempt to inline recursive functions and (2) it doesn't attempt inter-function inlining.

However, it attempts to cooperate with the lambda-lifter to remove all possible sites of partial applications, which are also heap allocated, to minimize memory allocation. The program shown in the previous section, undergoes the cycle in Fig 10

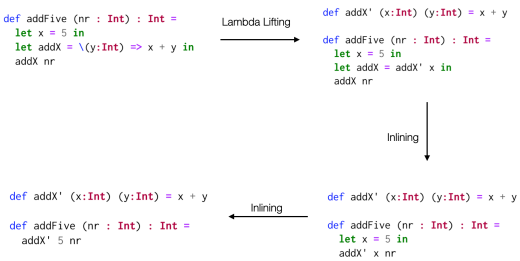


Figure 10: Lambda-lifting and inlining in action

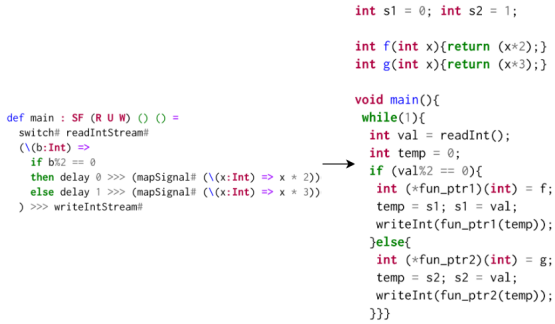


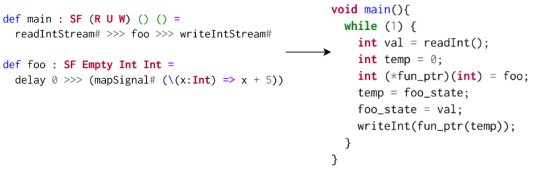
Figure 11: Code generation for switch#

The lambda lifting pass produces a partial application which is further inlined to a single function call where the arguments can be passed using registers. We show the generated LLVM code before and after the optimization passes are run in the extended version of this paper [55]. We show there the absence of any calls to `malloc` in the optimized version of the code. Our inliner doesn't employ any novel techniques but it still has to deal with engineering challenges like dealing with *name capture* [5], which it solves using techniques from the GHC inliner [34].

4.3 Code Generation

The Hailstorm compiler is designed as a linear pass through a tower of interpreters which compile away high level features, in the tradition of Reynolds [51]. Here, we show the final code generation for two of the more interesting combinators using C-like notation.

4.3.1 loop#. The `loop#` combinator models a traditional Mealy machine whose output depends on the input as well as the current state of the machine. In the following we see the code generated for the `delay` combinator described in Section 2.3 which is itself described using `loop#`.



In Erlang, the global variable of `foo_state` is modeled recursively using a global state map, which is updated on every time step. In the LLVM backend, the translation is very similar to the C-notation shown above.

4.3.2 switch#. We show an example of the `switch#` combinator when dealing with stateful branches

The second parameter in the `switch#` type - `(b→SF b c)` - acts like a macro which is unfolded into an if-then-else expression in the `L1` core language. The core language contains primops for *getting* and *setting* state variables. The final `L1` fragment that is generated for the `(b→SF b c)` component of `switch#` (post *partial evaluation* phase) is given below:

```
... -- s1 and s2 recursively in scope
let x = readInt() in
((λ val .
   if(val % 2 == 0)
   then (λ val2 .
          let temp = s1 in
          let _ = (set s1 to val2) in
          writeInt (f temp))
   else (λ val3 .
          let temp = s2 in
          let _ = (set s2 to val3) in
          writeInt (f temp))
  ) x) x ....
```

From the above `L1` fragment, the C like imperative code shown in Fig. 11 is generated. The if-then-else expressions are translated to case expressions in Erlang and the LLVM translation is very similar to the C code. The global state map now stores two state variables which are updated depending on the value of `x`. To make the operational aspect of the combinator clearer, we show the evolution of the state variables through the various timesteps of the program:

```
INPUT:  2  3  4  5  6  7  8...
s1:     0  2  2  4  4  6  6...
s2:     1  1  3  3  5  5  7...
OUTPUT: 0  3  4  9  8  15 12...
```

4.4 Pull Semantics

In this section we discuss the semantics of data consumption in our compiler. There exist two principle approaches (1) demand-driven pull and (2) data-driven push of data. As Hailstorm's signal function semantics assume a continuous streaming flow of data triggering the dataflow graph, our compiler adopts a pull based approach which continuously *polls* the I/O drivers for data. The program blocks until more data is available from the I/O drivers.

Let us take the earthquake detection example from Section 2.4. The *getSample* input function in the dataflow graph (Fig. 4) is a

wrapper around the driver for a simulated seismometer, which when polled for data provides a reading. The rightmost edge of the graph for the `Detection Event` pulls on the dataflow graph after it completes an action and the rest of the graph in turn pulls on `getSample`, which polls the simulated seismometer.

However, in certain devices such as UART, a push based model is more prevalent, where data is asynchronously pushed to the drivers. In such cases, to avoid dropping data the wrapper function (such as `getSample`) needs to be stateful and introduce buffers that store the data. The pseudocode of the wrapper function for such a driver, written for our Erlang backend, would look like Fig 12.

```
loop(State)
  receive
    {hailstormcall, From, Datasize} ->
        (Data, NewState) = extract_Data(Datasize, State),
        From ! {ok, Data},
        loop(NewState);
    {uartdriver, Message} ->
        NewState = buffer(Message, State),
        loop(NewState);
  end.
```

Figure 12: Enforcing pull semantics on push-based data

In Fig 12 there are two separate message calls handled. The data transmission from the drivers is handled using the `uartdriver` message call, which continuously buffers the data. On the other hand, the Hailstorm program, upon finishing one cycle of computation, requests more data using the `hailstormcall` message, and proceeds with the rest of the cycle.

4.4.1 Limitation. Elliott has criticized the use of pull semantics [19] as being wasteful in terms of the re-computation required in the dataflow graph. He advocates a hybrid *push-pull* approach, which, in case of continuously changing data, adopts the pull model, but in the absence of any change in the data doesn't trigger the dataflow graph. This approach could be useful in resource constrained devices, where energy consumption is an important parameter, and we hope to experiment with this approach in future work.

4.5 The digital - analog interface

The runtime of Hailstorm has to deal with the boundary of discrete digital systems and continuous analog devices. The input drivers have to frequently discretize events that occur at some unknown point of time into a stream of discrete data. An example is the Stopwatch simulation from Section 2.5.1. The pressing of an ON button in a stopwatch translates to the stream of ones (111...) and when switched OFF the simulation treats that as a stream of zeroes. This stream transformation is handled by the wrapper functions around the input drivers.

On the contrary, for the output drivers a reverse translation of discrete to continuous is necessary. We can take the example of operating traffic lights (related demonstration in Section 5.1.2). When operating any particular signal like GREEN supplying a discrete stream of data (even at the lowest granularity) will lead to a *flickering* quality of the light. In that case the wrapper function for the light drivers employs a stateful edge detector, as discussed in Section 2.4, to supply a new signal only in case of change.

4.6 Backend specific implementation

The backend implementation includes

- *Memory management.* The compiler attempts to minimize the amount of dynamically allocated memory using lambda-lifting and inlining such that the respective garbage collectors have to work less. Future work hopes to experiment with static memory management schemes like *regions* [61].
- *Tail call optimization (TCO).* Erlang itself does TCO and LLVM supports TCO when using the `fastcc` calling convention.

5 EVALUATION

5.1 Case Studies

In this section we demonstrate examples from the synchronous language literature [30] written in Hailstorm.

5.1.1 Watchdog process. A watchdog process monitors a sequential order processing system. It raises an alarm if processing an order takes more than a threshold time. It has two input signals - (1) *order* : *SF O* () *Bool* which emits *True* when an order is placed and *False* otherwise, (2) *done* : *SF D* () *Bool* which also emits *True* only when an order is done. For output we use - *alarm* : *SF A Bool* () where an alarm is rung only when *True* is supplied. In the program below we keep a threshold time for order processing as 3 seconds.

```
def f ((order : Bool, done : Bool),
       (time : Int, openOrder : Bool)) : (Bool, (Int, Bool))
 = if (openOrder == True && time > 3)
   then (True,(time + 1, False)) -- set alarm once
   else if (done  == True)
        then (False, (0, False)) -- reset
        else if (order  == True)
             then (False, (0, True))
             else (False, (time + 1, openOrder))

def watchdog : SF (O ∪ D ∪ A) () () = (order &&& done) >>>
  (loop# (0, False) (mapSignal# f)) >>> alarm

def main : SF (O ∪ D ∪ A) () () = rate# 1.0 watchdog
```

5.1.2 A simplified traffic light system. We take the classic example of a simplified traffic light system from the Lustre literature [53]. The system consists of two traffic lights, governing a junction of two (one-way) streets. In the default case, traffic light 1 is green, traffic light 2 is red. When a car is detected at traffic light 2, the system switches traffic light 1 to red, light 2 to green, waits for 20 seconds, and then switches back to the default situation.

We use a sensor - *sensor* : *SF S* () *Bool* - which keeps returning *True* as long as it detects a car. The system, upon detecting a subsequent car, resets the wait time to another 20 seconds. We sample from the sensor every second. For the traffic lights, we use 1 to indicate green and 0 for red.

```
def lightSwitcher (sig : Bool, time : Int):((Int, Int), Int)
  = if (time > 0)
    then ((0,1), time - 1)
    else if (sig == True)
         then ((0,1), 20) -- reset
         else ((1,0), 0) -- default

def lightController : SF (S ∪ TL1 ∪ TL2) () ((), ()) =
  sensor >>> (loop# 0 (mapSignal# lightSwitcher)) >>>
  (trafficLight1 *** trafficLight2)

def main : SF (S ∪ TL1 ∪ TL2) () () =
  rate# 1.0 lightController

-- sensor : SF S () Bool
-- trafficLight1 : SF TL1 Int ()
-- trafficLight2 : SF TL1 Int ()
```

5.1.3 *A railway level crossing.*

The problem. We consider a two-track railway level crossing area that is protected by barriers, that must be closed in time on the arrival of a train, on either track. They remain closed until all trains have left the area. The barriers must be closed 30 seconds before the expected time of arrival of a train. When the area becomes free, barriers could be opened, but it's not secure to open them for less than 15 s. So the controller must be warned 45 s before a train arrives. Since the speed of trains may be very different, this speed has to be measured, by detecting the train at two points separated by a known distance. A first detector is placed 2500 m before the crossing, and a second one 100 m after this first. A third is placed after the crossing area, and records a train's leaving. We divide our solution into three programs.

The Detect Process. The passage of a train is detected by a mechanical device, producing a *True* pulse - *pulse* : *SF P* () *Bool* - only when a wheel runs on it (otherwise *False*). The `detect` process receives all these pulses, but warns the controller only once, on the first wheel. All following pulses are ignored.

```
def f (curr : Bool, old : Bool) : (Bool, Bool) =
  if (curr == True && old == False)
  then (True, curr) else (False, curr)

def detect : SF P () Bool =
  pulse >>> loop# False (mapSignal# f)
```

A Track Controller. On each track, a controller receives signals from two detectors. From `Detect1` and `Detect2`, it must compute the train's speed, and warn the barriers 45 seconds before expected time of arrival at crossing. At the maximum speed of 180 km/h, the 100 m between `Detect1` and `Detect2` are covered in 2 s. So, a clock pulse every 0.1 s would be of good accuracy.

```
def t ((d1 : Bool, d2 : Bool), time:Float) : (Float, Float)
 = case (d1, d2) of
     (True, False) ~> (0.0,0.0);
     (False, True) ~> ((24.0 *. (time +. 0.1) -. 45.0),0.0);
     _             ~> (0.0, time +. 0.1)

def timer : SF Empty (Bool, Bool) Float
 = rate# 0.1 (loop# 0.0 (mapSignal# t))

def trackController : SF (P1 ∪ P2) () Float
 = (detect1 &&& detect2) >>> timer

def detect1 : SF P1 () Bool = ...
def detect2 : SF P2 () Bool = ...
```

When a train approaches, the `trackController` calculates the time for the train to reach the barrier and sends that value. In the absence of a train it sends zeroes.

The Barriers Controller. It consists of an alarm - *alarm* : *SF* ($P_1 \cup P_2$) () *Bool* which consumes the time values from the `trackController` and returns *True* when an alarm is to be rung. The `trackController` is also sampled every 0.1 second.

```
def g (sig : Float, time : Float) : (Bool, Float) =
  if (sig > 0.0)
  then (False, sig)
  else if (time == 0.1)
       then (True, 0.0)
       else (False, time -. 0.1)

def alarm : SF (P1 ∪ P2) () Bool
  = trackController >>> rate# 0.1 (loop# 0.0 (mapSignal# g))
```

Given the alarms from the two separate tracks, the barrier controller sends an `open/close` signal represented by 0 and 1 respectively. In case a train is approaching in both of the tracks at the same speed - the barrier for only track 1 is opened.

```
def alarm1 : SF (P1 ∪ P2) () Bool = ...
def alarm2 : SF (P3 ∪ P4) () Bool = ...

def openclose (sig : (Bool, Bool)) : (Int, Int) =
  case sig of
     (True, False)  ~> (0, 1);
     (False, True)  ~> (1, 0);
     (True , True)  ~> (0, 1);
     _  ~> (0, 0) -- (False, False)

def barrierController : SF (P1 ∪ P2 ∪ P3 ∪ P4) () (Int, Int)=
    alarm1 &&& alarm2 >>> (mapSignal# openclose)
```

Finally, before sending the signal to the actuators (i.e the barriers), we need an additional system clock that keeps each barrier open for 45 seconds, and ignores other signals in the interim. The handling of the conversion of discrete signals to continuous is done by the drivers for the actuators, as discussed in Section 4.5.

```
def gate ((x : Int, y : Int),
          (t : Float, old : (Int, Int))) :
         ((Int, Int) , (Float, (Int, Int))) =
  if (old ≠ (0,0) ∧ t > 0.0)
  then (old, ((t -. 0.1), old)) -- persisting a signal
  else if (x == 1 ∨ y == 1)
       then ((x,y), (45.0, (x, y)))
       else ((x,y), (0.0, (x,y)))

def main : SF (P1 ∪ P2 ∪ P3 ∪ P4 ∪ B1 ∪ B2) () ((), ()) =
    rate# 0.1 (barrierController >>>
               (loop# (0.0, (0,0)) (mapSignal# gate)) >>>
               (barrier1 *** barrier2))
```

```
-- barrier1 : SF B1 Int ()
-- barrier2 : SF B2 Int ()
```

Note that the three instances of `rate#` all sample at the same interval of 0.1 seconds. Hailstorm currently doesn't have well defined semantics for programs with multiple clock rates.

5.2 Microbenchmarks

Here we provide memory consumption and response-time microbenchmarks for the examples presented above using the Erlang backend.

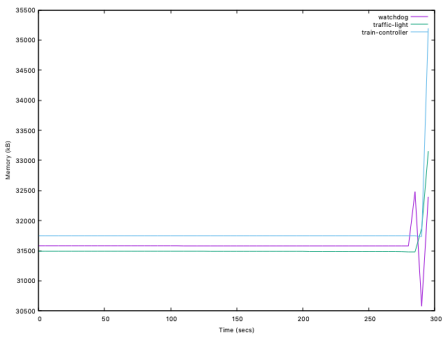


Figure 13: Memory consumption of programs

We measure the mean memory consumption for each program over five runs, each of five minutes duration. Given the I/O driven nature of the programs, the memory consumption shows little to no fluctuations. The Erlang runtime (ERTS) upon initialization sets up the garbage collector, initializes the lookup table and sets up the bytecode-interpreter which occupies 30 MB of memory on average. The actual program and its associated bookkeeping structures takes up an average of 1.5 MB of memory in the programs above. The garbage collector remains inactive throughout the program run. The memory spike visible upon termination is the garbage collector pausing the program and collecting all residual memory.

Program	Run_1(ms)	Run_2(ms)	Run_3(ms)
watchdog	7.7	8.65	11.4
traffic-light	3.81	3.04	2.12
train controller	29.72	28.05	29.8

Table 1: Response time measured in milliseconds

Table 1 shows the response time for the programs measured in milliseconds. We measure the CPU Kernel time (CPUT) - which calculates the time taken by the dataflow graph to finish one cycle of computation. We show three separate runs where the Erlang virtual machine is killed and restarted to reset the garbage collector. Each of the numbers are an average of twenty iterations of data processing. We use the `erlang:statistics` module for measuring time and in the applications I/O happens over the command line interface, which explains the overall slow behaviour (tens of milliseconds). The metrics are run on the `erts-10.6.4` runtime and virtual machine running on a Macbook-Pro with a 2.9 GHz Intel Core i9 processor. A common observation is that the computation takes less than 1% of the total wall clock time involved in the response rate, showing that the I/O reading/writing times dominate the final response rate.

An alternate benchmarking strategy which we used was to model the input from the sensors as an in-memory structure (in Erlang) and compute the total response time for processing those values using the `timer:tc` module:

Program	$Run_1(\mu s)$	$Run_2(\mu s)$	$Run_3(\mu s)$
watchdog	97.3	106.7	98.7
traffic-light	110.8	120.2	115.1
train controller	144.3	128.1	138.7

The above values are all measured in microseconds which are averaged over forty iterations each. As expected, using an in-memory structure results in graph processing times that are much lower than those in Table 1.

6 RELATED WORK

6.1 Programming Languages for IoT

There has been recent work on designing embedded DSLs (EDSLs) for IoT applications [11]. In EDSLs, I/O is handled by embedding a pure core language inside a host language's I/O model - which is an approach that Hailstorm deliberately avoids. Given the I/O dominated nature of IoT apps, we choose to focus much of our attention on designing a composable stream based I/O model, rather than only considering the pure core language, as many EDSLs do.

Other approaches like Velox VM [62] runs general purpose languages like Scheme on specialized virtual machines for IoT devices. A separate line of work has been exploring restrictive, Turing-incomplete, rule-based languages like IoTDSL [2] and CyprIOT [9].

Juniper [29] is one of the few dedicated languages for IoT but it exclusively targets Arduino boards. Emfrp [56] and its successor XFRP [59] are most closely related to the goals of Hailstorm. However, their model of I/O involves writing glue code in C/C++ and embedding the pure functional language inside it. Hailstorm has a more sophisticated I/O integration in the language.

While IoT stands for an umbrella term for a large collection of software areas, there has been research on *declarative* languages for older and *specialized* application areas like:

- *Wireless Sensor Networks (WSNs).* There exists EDSLs like Flask [42] and macroprogramming languages like Regiment [45] and Kairos [24] for WSNs.
- *Real Time Systems.* Synchronous language like Esterel [10], Lustre [26] are a restrictive set of languages designed specifically for real time systems. Further extensions of these languages like Lucid Synchrone [12], ReactiveML [44], Lucy-n [43] have attempted to makes them more expressive.

The applications demonstrated in the paper are expressible in synchronous languages, albeit using a very different interface from Hailstorm. While languages like Lustre and its extensions are *pure*

they restrict their synchronous calculus to the pure core language and handle I/O using the old stream based I/O model of Haskell [48]. Hailstorm explores the design space of *pure* functional programming with the programming model and purity encompassing the I/O parts as well.

In Lustre, a type system called the clock calculus ensures that programs can run without any implicit buffering inside the program. Strong safety properties such as determinism and absence of deadlock are ensured at compile time, and programs are compiled into statically scheduled executable code. This comes at the price of reduced flexibility compared to synchronous dataflow-like systems, particularly in the ease with which bounded buffers can be introduced and used. Mandel et al have studied n-synchronous systems in an attempt to bring greater flexibility to synchronous languages [43]. Hailstorm, in the presence of recursion, is unable to statically predict memory usage but we plan future work on type level encoding of buffer sizes to make memory usage more predictable. *Polychronous* languages like SIGNAL [8] and FRP libraries like Rhine [6] provide static guarantees on correctness of systems with multiple clocks - something that we hope to experiment with in the future.

6.2 FRP

Hailstorm draws influence from the FRP programming model. Since the original FRP paper [18], it has seen extensive research over various formulations like arrowized FRP [46], asynchronous FRP [16], higher-order FRP [35], monadic stream functions [47].

Various implementations have explored the choice between a static structure of the dataflow graph (for example Elm [15]) or dynamic structure, as in most Haskell FRP libraries [3] as well as FrTime in Racket [13]. The dynamic graph structure makes the language/library more expressive, allowing programs like sieves [25].

The higher-order FRP implementation offers almost the local maxima of tradeoffs, but at the cost of an extremely sophisticated type system, which infests into the source language, compromising its simplicity.

The `loop#` combinator in Hailstorm is similar to the μ-combinator first introduced by Sheeran [57] and to the `loopB` combinator in the Causal Commutative Arrows (CCA) paper [40]. CCA presents a number of mathematical laws on arrows and utilizes them to compile away intermediate structures and generate efficient FRP code - a promising avenue for future work in Hailstorm.

FRP has seen adoption in various application areas. Application areas related to our target areas of IoT applications include robotics [64], real-time systems [63], vehicle simulations [21] and DSLs discussed in detail in the previous section.

6.3 Functional I/O

A detailed summary of approaches to functional I/O was presented by Gordon et al. [22]. Since then monadic I/O [23] has become the standard norm for I/O in pure functional languages, with the exception of Clean's I/O system [1] based on uniqueness types.

More recently, there have been attempts at non mondaic I/O using a state passing trick in the Universe framework [20]. The latest work has been on the notion of resource types, proposed by Winograd-Cort et al. [66], which is explored as a library in Haskell. In effect, their library uses Haskell's monadic I/O model; however, they mention possible future work on designing a dedicated language for resource types. Hailstorm explores that possibility by integrating the idea of resource types natively in a language's I/O model. FRPNow! [49] provides an alternate monadic approach to integrate I/O with FRP.

7 FUTURE WORK

Hailstorm is an ongoing work to run a pure, statically-typed, functional programming language on memory constrained edge devices. As such, there are a number of open avenues for research:

- *Security.* We hope to integrate support for Information Flow Control (IFC) [28] - which uses language based privacy policies to determine safe dataflow - in Hailstorm. Given that the effectful combinators are based on the Arrow framework, we expect to build on two lines of work - (1) integrating IFC with the Arrow typeclass in Haskell [37] (2) a typeclass based technique to integrate IFC without modifying the compiler [52] but one which relies on the purity and static typing of the language.

- *Reliability.* Hailstorm currently doesn't provide any fault tolerance strategies to mitigate various node/communication failures. However, being hosted on the Erlang backend, we plan to experiment with distributed versions of the arrow combinators where the underlying runtime would use *supervision trees* to handle failures. This would be a much more invasive change as the synchronous dataflow model of the language is not practically suitable in a distributed scenario - leading to more interesting research directions on *macro-programming models* [24].

- *Memory constrained devices.* The evaluation of this paper is carried out on GRiSP boards which are relatively powerful boards. We are currently working on developing a small virtual machine which can interpret a functional bytecode instruction set and run on much more memory constrained devices like STM32 microcontrollers.

8 CONCLUSION

We have presented the design and implementation of Hailstorm, a domain specific language targeting IoT applications. Our evaluation suggests that Hailstorm can be used to declaratively program moderately complex applications in a concise and safe manner. In the future, we hope to use the purity and type system of Hailstorm to enforce language-based security constraints, as well as to increase its expressiveness to enable the description of interacting IoT devices in a distributed system.

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023). We would also like to thank Henrik Nilsson and Joel Svensson for their valuable feedback on improving our paper.

REFERENCES

[1] Peter Achten and Rinus Plasmeijer. 1995. The ins and outs of Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.
[2] Moussa Amrani, Fabian Gilson, Abdelmounaim Debieche, and Vincent Englebert. 2017. Towards User-centric DSLs to Manage IoT Systems.. In *MODELSWARD*. 569–576.
[3] Edward Amsden. 2011. A survey of functional reactive programming. *Rochester Institute of Technology* (2011).
[4] Joe Armstrong. 1997. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 196–203.

[5] Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

[6] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 145–157. https://doi.org/10.1145/3242744.3242757

[7] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If this then what? Controlling flows in IoT apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1102–1119.

[8] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming* 16, 2 (1991), 103–149.

[9] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. 2019. CyprIoT: framework for modelling and controlling network-based IoT applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 832–841.

[10] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

[11] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT modules as a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. 15–20.

[12] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous Functional Programming: The Lucid Synchrone Experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008).

[13] Gregory H Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*. Springer, 294–308.

[14] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 7–18.

[15] Evan Czaplicki. 2012. Elm: Concurrent Frp for Functional GUIs. *Senior thesis, Harvard University* (2012).

[16] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices* 48, 6 (2013), 411–422.

[17] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.

[18] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 263–273.

[19] Conal M Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. 25–36.

[20] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A functional I/O system or, fun for freshman kids. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 47–58. https://doi.org/10.1145/1596550.1596561

[21] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*. 43–47.

[22] Andrew Donald Gordon. 1992. *Functional programming and input/output*. Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259479

[23] Andrew D Gordon and Kevin Hammond. 1995. Monadic I/O in Haskell 1.3. In *Proceedings of the haskell Workshop*. 50–69.

[24] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems*. Springer, 126–140.

[25] Heine Halberstam and Hans Egon Richert. 2013. *Sieve methods*. Courier Corporation.

[26] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[27] Eric Haugh and Matt Bishop. 2003. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2003/testing-c-programs-buffer-overflow-vulnerabilities/

[28] Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. *Software safety and security* 33 (2012), 319–347.

[29] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. 8–16.

[30] Bernard Houssais. 2002. The synchronous programming language SIGNAL: A tutorial. *IRISA, April* (2002).

[31] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

[32] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*. Springer, 190–203.

[33] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

[34] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming* 12, 4-5 (2002), 393–434.

[35] Neelakantan R Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. *ACM SIGPLAN Notices* 48, 9 (2013), 221–232.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[37] Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical computer science* 411, 19 (2010), 1974–1994.

[38] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. 2017. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal* 4, 5 (2017), 1125–1142.

[39] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2010. The arrow calculus. *Journal of Functional Programming* 20, 1 (2010), 51–69.

[40] Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. *ACM Sigplan Notices* 44, 9 (2009), 35–46.

[41] Hai Liu and Paul Hudak. 2007. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science* 193 (2007), 29–45.

[42] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 335–346.

[43] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-synchronous extension of Lustre. In *International Conference on Mathematics of Program Construction*. Springer, 288–309.

[44] Louis Mandel and Marc Pouzet. 2005. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 82–93.

[45] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. In *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE, 489–498.

[46] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 51–64.

[47] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. *ACM SIGPLAN Notices* 51, 12 (2016), 33–44.

[48] Simon Peyton Jones. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction* 180 (2001), 47.

[49] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 302–314.

[50] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 421–434.

[51] John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. 717–740.

[52] Alejandro Russo. 2015. Functional pearl: two can keep a secret, if one of them uses Haskell. *ACM SIGPLAN Notices* 50, 9 (2015), 280–288.

[53] Philipp Rümmer. 2014 (accessed May 13, 2020). *An Introduction to Lustre*. http://www.it.uu.se/edu/course/homepage/modbasutv/ht14/Lectures/lustre_slides_141006.pdf

[54] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[55] Abhiroop Sarkar. 2020. Hailstorm - A Statically-Typed, Purely Functional Language for IoT Applications. http://abhiroop.github.io/pubs/hailstorm.pdf. [Online; accessed 19-July-2020].

[56] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*. 36–44.

[57] Mary Sheeran. 1984. muFP, A Language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 104–112.

[58] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[59] Kazuhiro Shibanai and Takuo Watanabe. 2018. 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*. 13–22.

[60] Peer Stritzinger. (accessed May 13, 2020). *GRiSP embedded system boards*. https://www.grisp.org/

[61] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.
[62] Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. *Journal of Network and Computer Applications* 118 (2018), 61–73.
[63] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 146–156.
[64] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 155–172.
[65] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. 2005. Monitoring volcanic eruptions with a wireless sensor network. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.* IEEE, 108–120.
[66] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. 2012. Virtualizing real-world objects in FRP. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 227–241.

PART 2

Chapter 3

Higher-Order Concurrency for Microcontrollers

Higher-Order Concurrency for Microcontrollers

Abhiroop Sarkar
Robert Krook
Bo Joel Svensson
Mary Sheeran
Chalmers University
Gothenburg, Sweden
{sarkara,krookr,joels,mary.sheeran}@chalmers.se

Abstract

Programming microcontrollers involves low level interfacing with hardware and peripherals that are concurrent and reactive. Such programs are typically written in a mixture of C and assembly using concurrent language extensions (like `FreeRTOS tasks` and `semaphores`), resulting in unsafe, callback-driven, error-prone and difficult-to-maintain code.

We address this challenge by introducing `SenseVM` - a bytecode-interpreted virtual machine that provides a message passing based *higher-order concurrency* model, originally introduced by Reppy, for microcontroller programming. This model treats synchronous operations as first-class values (called `Events`) akin to the treatment of first-class functions in functional languages. This primarily allows the programmer to compose and tailor their own concurrency abstractions and, additionally, abstracts away unsafe memory operations, common in shared-memory concurrency models, thereby making microcontroller programs safer, composable and easier-to-maintain.

Our VM is made portable via a low-level *bridge* interface, built atop the embedded OS - Zephyr. The bridge is implemented by all drivers and designed such that programming in response to a software message or a hardware interrupt remains uniform and indistinguishable. In this paper we demonstrate the features of our VM through an example, written in a Caml-like functional language, running on the `nRF52840` and `STM32F4` microcontrollers.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Runtime environments**; **Functional languages**; **Interpreters**.

MPLR '21, September 29–30, 2021, Münster, Germany

ACM ISBN 978-1-4503-8675-3/21/09…$15.00
https://doi.org/10.1145/3475738.3480716

Keywords: concurrency, virtual machine, microcontrollers, functional programming

ACM Reference Format:
Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, and Mary Sheeran. 2021. Higher-Order Concurrency for Microcontrollers. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '21), September 29–30, 2021, Münster, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3475738.3480716

1 Introduction

Microcontrollers are ubiquitous in embedded systems and IoT applications. These applications, like robot controllers, cars, industrial machinery, are inherently concurrent and event-driven (termed *reactive* in more recent literature). A 2011 poll[26] conducted among embedded systems developers found `C`, `C++` and `assembly` to be the most popular choices for programming such reactive applications.

The popularity of the `C` family can be attributed to its fine-grained control over memory layout and management. Also, `C` compilers are extremely portable across a diverse set of microcontrollers while offering low-level control over hardware peripherals. However, `C` not being an innately concurrent language, embedded OSes like `FreeRTOS` and `Zephyr` provide their own shared memory concurrency abstractions like threads, semaphores, mutexes etc [32–34]. Additionally, the event-driven driver interfaces in such OSes tend to have APIs that look like the following:

```
int gpio_pin_interrupt_configure(const struct device *port
                                 , gpio_pin_t pin
                                 , gpio_flags_t flags);
void gpio_init_callback(struct gpio_callback *callback
                        , gpio_callback_handler_t handler
                        , gpio_port_pins_t pin_mask);
int gpio_add_callback(const struct device *port
                      , struct gpio_callback *callback);
```

This combination of programming in a memory-unsafe language, like `C`, callback-based driver APIs and shared-memory concurrency primitives leads to error-prone, difficult-to-maintain and unsafe programs. Moreover, programs also end up being very difficult to port to other microcontroller boards and follow intricate locking protocols

and complex state machines to deal with the concurrent and reactive nature of the applications.

Recently, there has been a surge in dynamic-language-based runtime environments like MicroPython [9] and Espruino [31] on microcontrollers. While these languages abstract away the unsafe memory management of the C family, neither of them is inherently concurrent. Programming with the hardware peripherals in MicroPython has the following form:

```
def callback(x):
#...callback body with nested callbacks...
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING
                    , pyb.Pin.PULL_UP, callback)
ExtInt.enable()
```

The above is plainly a wrapper over the original C API and is prone to suffer from the additional and *accidental* complexity of nested callback programming, colloquially termed as *callback-hell*[18].

Contributions. In this paper we simplify the handling of callback-driven APIs, discussed above, by introducing a bytecode-interpreted virtual machine, SenseVM[1], which models all hardware and I/O interactions via a message-passing interface. We enumerate the practical contributions of SenseVM below:

1. **Higher-order concurrency**. We provide, to the best of our knowledge, the first implementation of the higher-order concurrency model [20] for programming microcontrollers. This model allows the introduction of first class values, called `Events`, for representing synchronous operations and provides combinators to compose more complex `Event` trees, which are useful for control-flow heavy programs, common in microcontrollers. We briefly summarize the model in Section 2.1 and describe implementation details in Section 3.
2. **Message-passing based I/O**. Noting that complex state machines and callback-hell arises from the intertwined nature of *concurrency* and *callback-based I/O*, we mitigate the issues by unifying concurrency with I/O. As a result, programming in response to a hardware interrupt or any other I/O message in SenseVM remains indistinguishable from responding to a software message. Moreover, owing to the higher-order concurrency model, the programs do not reduce to a chain of *switch-casing* of message contents, common in other message passing languages like Erlang. We explain implementation details of the message passing (Section 3.2), show a sample program (Section 2.3) and evaluate the performance metrics of this I/O model (Section 4).
3. **Portability**. Portability among microcontroller boards is challenging, as a consequence of the diverse set of peripherals and hardware interfaces available. To address this, SenseVM provides a *low-level bridge* interface written in `C99` (described in Section 3.3) which provides a common API which can be implemented by various synchronous and asynchronous drivers. Once drivers of different boards implement this interface, programs can be trivially ported between them as we demonstrate by running the same, unaltered program on the `nRF52840` and `STM32F4` based boards.

[1] https://github.com/svenssonjoel/Sense-VM

2 Programming on SenseVM

To demonstrate programming on top of the SenseVM, we will use an eagerly-evaluated, statically-typed, functional language which extends the polymorphic lambda calculus with `let` and `letrec` expressions similar to Caml [16]. Unlike Caml, it lacks the mutable `ref` type and does all I/O operations via the message-passing interface of the VM. Type declarations and signatures in our language are syntactically similar to those of Haskell [14].

The message-passing interface of SenseVM is exposed via runtime supplied functions. It is *synchronous* in nature and all communications happen over *typed channels*. In a Haskell-like notation, the general type signature of message sending and receiving over channels could be written:

```
sendMsg : Channel a -> a -> ()
recvMsg : Channel a -> a
-- a  denotes any generic type like Int, Bool etc;
-- () denotes the "void" type
```

However, our VM implements an extension of the above known as *higher-order concurrency* [20]. We describe the differences and their implications on the programming model in the following section.

2.1 Higher-Order Concurrency

The central idea of *higher-order concurrency* is to separate the act of synchronous communication into two steps -

1. Expressing the intent of communication
2. Synchronisation between the sender and the receiver

The first step above produces first-class values called `Events`, which are concrete runtime values, provided by the SenseVM. The second step, synchronisation, is expressed using an operation called `sync`. Now we can write the type signature of message passing in our VM as:

```
send : Channel a -> a -> Event ()
recv : Channel a -> Event a
sync : Event   a -> a
```

Intuitively, we can draw an equivalence between general message passing and higher-order concurrency based message passing, using function composition, like the following:

```
sync . send ≡ sendMsg
sync . recv ≡ recvMsg
```

The above treatment of "Events as values" is analogous to the treatment of "functions as values" in functional programming. In a similar spirit as higher-order functions, our VM provides Event based combinators for further composition of trees of Events:

```
choose : Event a -> Event a  -> Event a
wrap   : Event a -> (a -> b) -> Event b
```

The choose operator represents the standard *selective communication* mechanism, necessary for threads to communicate with multiple partners, found in CSP [12]. The wrap combinator is used to run *post-synchronisation* operations. wrap ev f can be read as - "post synchronisation of the event ev, apply the function f to the result".

Reppy draws parallels between an Event and its associated combinators with higher-order functions [21], using the following table:

Property	Function values	Event values
Type constructor	->	event
Introduction	λ - abstraction	receive send etc.
Elimination	application	sync
Combinators	composition map etc.	choose wrap etc.

SenseVM provides other combinators like spawn for spawning a *lightweight* thread and channel for creating a typed channel:

```
spawn   : (() -> ()) -> ThreadId
channel : ()  -> Channel a
```

Next we discuss the message-passing API for handling I/O.

2.2 I/O in SenseVM

The runtime APIs, introduced in the previous section, are useful for implementing a software message-passing framework. However, to model external hardware events, like interrupts, we introduce another runtime function:

```
spawnExternal : Channel a -> Int -> ThreadId
```

This function connects the various peripherals on a microcontroller board with the running program, via typed channels. The second argument to spawnExternal is a *driver-specific identifier* to identify the driver that we wish to communicate with. Currently in our runtime, we use a monotonically increasing function to number all the drivers starting from 0; the programmer uses this number in spawnExternal. However, as future work, we are building a tool that will parse a configuration file describing the drivers present on a board, number the drivers, inform the VM about the numbering and then generate a frontend program like the following:

```
data Driver = LED Int | Button Int | ...

led0 = LED 0
led1 = LED 1
but0 = Button 2
but1 = Button 3

--Revised `spawnExternal` type signature will be
spawnExternal : Channel a -> Driver -> ThreadId
```

In the rest of the paper, we will be referring to the revised spawnExternal function for clarity. Now we can express a SenseVM program that listens to an interrupt raised by a button press below:

```
main =
  let bchan = channel() in
  let _ = spawnExternal bchan but0 in
  sync (recv bchan)
```

The recv is a *blocking* receive and if there are other processes that can be scheduled while this part of the program is blocked, the runtime will schedule them. In the absence of any other processes, the runtime will relinquish its *control* to the underlying OS (Zephyr OS) and will **never poll** for the button press. The SenseVM runtime is geared towards IoT applications where microcontrollers are predominantly asleep and are woken up reactively by hardware interrupts. In the following section, we demonstrate a more complete program running on our VM.

2.3 Button-Blinky

Next we *portably* run a program in both the nRF52840 and STM32F4 microcontroller based boards. The program *indefinitely* waits for a button press on button 0 and on receiving an ON signal, sends an ON signal to LED number 0. Upon the button release, it receives an OFF signal and sends the same to the LED. The LED stays ON as long as the button is pressed.

This program, expressed in C and hosted in Zephyr, is about 127 lines of code (see [7]) involving setup, initialisation, callback registration and other control logic. The same program expressed on top of the SenseVM is:

```
1  bchan = channel ()
2  lchan = channel ()
3
4  main =
5    let _ = spawnExternal bchan but0 in
6    let _ = spawnExternal lchan led0 in
7    buttonBlinky
8
9  buttonBlinky =
10   let _ = sync (wrap (recv bchan) blinkled) in
11   buttonBlinky
12   where
13     blinkled i = sync (send lchan i)
```

In the above program, we create a channel per driver (Line no. 1 and 2) and instruct the button driver, via `spawnExternal`, to send any hardware interrupts to the registered channel - `bchan` (Line no. 5). Upon receiving an interrupt, we run a post-synchronisation action using `wrap` (Line no. 10), which sends the value sent by the interrupt to the LED driver using `lchan` (Line no. 13). It recursively calls itself to continue running the program infinitely (Line no. 11). There is a notable absence of callbacks in the above program.

3 Design and Implementation

3.1 System Overview

SenseVM, including its execution unit, is an implementation of the Categorical Abstract Machine [5] (CAM), as explained by Hinze [11], but has been augmented with a set of operation codes for the higher-order concurrency extensions. We show the compilation and runtime pipeline of the VM in Figure 1 below:

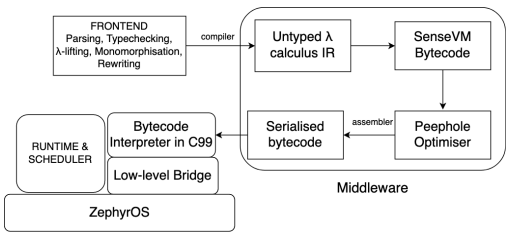


Figure 1. The SenseVM compilation & runtime pipeline

Frontend. The frontend supports a polymorphic and statically typed functional language whose types are monomorphised at compile time. It supports some optimisation passes like lambda-lifting to reduce heap allocation.

Middleware. The frontend's desugared intermediate representation is compiled to an untyped lambda calculus representation. This representation is further optimised by generation of specialised bytecodes for an operational notion called *r-free* variables, as described by Hinze[11], to further reduce heap allocation. The generated SenseVM bytecode operates on a stack machine with a single environment register. The bytecode is then further subjected to peephole optimisations like β–reduction and last-call optimisation[11] (a generalisation of tail-call elimination).

Backend. The SenseVM back-end, or virtual machine, is split into a high-level part and a low-level part. Currently the low-level part is implemented on top of Zephyr and is described in more detail in Section 3.3. The interface between the low-level, Zephyr based, part of the back-end and the high-level has been kept minimal to enable plugging in other embedded OSes like `FreeRTOS`.

The high-level part of the back-end consists of a fixed number of *contexts*. A context is a lightweight thread comprising of (1) an environment register, (2) a stack and (3) a program counter. The context switching is *cooperatively* handled by the VM scheduler. The high-level part of the VM also contains a garbage-collected heap where all compound CAM values (like tuples) are allocated.

The VM uses a mark-and-sweep garbage collection algorithm that is a combination of the Hughes lazy sweep algorithm and the Deutsch-Schorr-Waite pointer-reversing marking algorithm [13, 15, 23]. As future work, we intend to investigate more static memory management schemes like regions [27] and also real-time GC algorithms[17]. The following section explains the message-passing based concurrency aspects of the VM.

3.2 Synchronous Message Passing

In the higher-order concurrency model of synchronous message passing, we separate the *synchronisation* from the *description* of message passing. By doing this, we introduce an intermediate value type known as an `Event`.

3.2.1 Event. Reppy calls `send` and `recv` operations *base-event* constructors. Operations like `choose` and `wrap` are the higher-order operators used for composing the base events. An `Event` is a concrete runtime value represented in the SenseVM heap as a nested tuple. Figure 2 shows the representation of an `Event` on the heap.

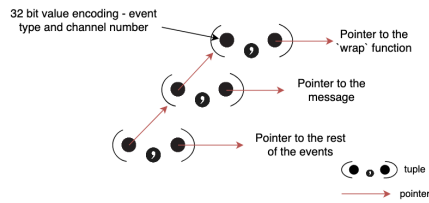


Figure 2. The heap structure of an Event

An `Event` is represented as a linked list of base events. We represent the linked list as a chain of tuples where the second element points to the rest of list, and the first element points to another nested tuple, which encodes - (1) message content, (2) channel number, (3) event type (`send` or `recv`) and (4) post-synchronisation function to be applied.

A composition operation like `choose e1 e2` simply appends two lists - `e1` and `e2`. We can further compose `choose` using `fold` operations[2]to allow it to accept a list of events and build complex event trees, along with `wrap`, as shown in the following program. However it is always possible to rewrite this tree at compile time, using function composition,

[2]https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#v:foldr1

to produce the canonical representation of events as a linked list rather than a tree.

```
choose' : [Event a] -> Event a
choose' = foldr1 choose

choose' [wrap bev1 w1,
         wrap (choose' [wrap bev2 w2,
                        wrap bev3 w3]) w4]
-- Rewritten to
choose' [ wrap bev1 w1,
          wrap bev2 (w4 . w2),
          wrap bev3 (w4 . w3) ]
```

3.2.2 Synchronisation. The `sync` or synchronisation operation is one of the more intricate aspects of the SenseVM runtime and we describe the algorithm in pseudocode below:

```
Function sync (eventList)
ev ← findSynchronisableEvent(eventList)
if ev ≠ ∅ then
  syncNow(ev)
else
  block(eventList)
  dispatchNewThread()
end if
EndFunction
```

The above gives a bird's-eye view of the major operations involved in `sync`. To understand the `findSynchronisableEvent` function, we should understand the structure of a *channel*, that contains a send and a receive queue. These queues are used not to hold the messages but to track which threads are interested in sending or receiving on the respective channel. Now we can describe `findSynchronisableEvent` :

```
Function findSynchronisableEvent (eventList)
for all e ∈ eventList do
  if e.channelNo communicating with driver then
    if llBridge.driver readable/writeable? then
      return e
    end if
  else
    if e.baseEventType == SEND then
      if ¬isEmpty(e.channelNo.recvq) then
        return e
      end if
    else if e.baseEventType == RECV then
      if ¬isEmpty(e.channelNo.sendq) then
        return e
      end if
    end if
  end if
end for
return ∅
EndFunction
```

When no synchronisable event is found we use `block` :

```
Function block (eventList)
for all e ∈ eventList do
  if e.baseEventType == SEND then
    enqueue(e.channelNo.sendq)
  else if e.baseEventType == RECV then
    enqueue(e.channelNo.recvq)
  end if
end for
EndFunction
```

After the call to `block`, we call `dispatch` described below:

```
Function dispatchNewThread ()
if readyQ ≠ ∅ then
  threadId ← dequeue(readyQ)
  currentThread = threadId
else
  relinquish control to Zephyr
end if
EndFunction
```

On receiving a synchronisable event, we apply `syncNow` :

```
Function syncNow (event)
if ¬hardware(event) then
  if event.baseEventType == SEND then
    threadIdR ← dequeue(event.channelNo.recvq)
    recvEvt ← threadIdR.envRegister
    event.Message → threadIdR.envRegister
    threadIdR.programCounter → recvEvt.wrapFunc

    currentThread.programCounter → event.wrapFunc

    sendingThread = currentThread
    currentThread = threadIdR
    schedule(sendingThread)
  else if event.baseEventType == RECV then
    threadIdS ← dequeue(event.channelNo.sendq)
    sendEvt ← threadIdS.envRegister
    sendEvent.Message → currentThread.envRegister

    threadIdS.programCounter → sendEvt.wrapFunc

    currentThread.programCounter → event.wrapFunc

    schedule(threadIdS)
  end if
else if hardware(event) then
  if event.baseEventType == SEND then
    llBridge.write(event.Message) → driver
  else if event.baseEventType == RECV then
    currentThread.envRegister ← llBridge.read(driver)
  end if
  currentThread.programCounter → event.wrapFunc
end if
EndFunction
```

In the case of interrupts that arrive when the SenseVM runtime has relinquished control to Zephyr, we place the message on the Zephyr queue and request the scheduler to wake up, create an event and then call sync on that event. For events in a choose clause that fail to synchronise, we clear them from the heap using the *dirty-flag* technique invented by Ramsey [19].

3.2.3 Comparison with Actors. In SenseVM, we opt for a synchronous style of message-passing, as distinct from the more popular asynchronous style found in actor-based systems[10]. Our choice is governed by the following:
(1) Asynchronous send (as found in actors) implies the *unboundedness* of an actor mailbox, which is a poor assumption in memory-constrained microcontrollers. With a bounded mailbox, actors eventually resort to synchronous send semantics, as found in SenseVM.
(2) Synchronous message passing can emulate buffering by introducing intermediate processes, which forces the programmer to think upfront about the cost of buffers.
(3) Acknowledgement becomes an additional, explicit step in asynchronous communication, leading to code bloat. Acknowledgement is implicit in the synchronous message passing model and is a practical choice when not communicating across distributed systems.
(4) Actor based systems (like Medusa [3]) incur the extra cost of tagging a message to identify which entities have requested which message. The combination of *channels* and synchronous message-passing ensures that an arriving hardware interrupt knows where to send the message, without any message identification cost.

3.3 Low-Level Bridge

The *low-level bridge* exists to bridge the divide in abstraction level from the channel-based communication of SenseVM to the underlying embedded OS - Zephyr [1]. The main motivation behind using the Zephyr RTOS as the lowest level hardware abstraction layer (HAL) for SenseVM is that it provides a good set of driver abstractions that are portable between microcontrollers and development boards. Examples of such drivers include UART, SPI, I2C, buttons and LEDs and many other peripherals.

A driver can be be either synchronous or asynchronous in nature. An example of a synchronous driver is an LED that can be directly read or written. On the other hand, asynchronous peripherals operate using interrupts. An asynchronous driver interrupt can be signaling, for example, that it has filled up a buffer in memory using Direct Memory Access (DMA) that now needs handling, or it could signal a simple boolean message such as "a button has been pressed".

To accommodate these various types of drivers, the connection between SenseVM and the drivers is made using a datatype called `ll_driver_t`, which describes an interface that each low-level (platform and HAL dependent) driver should implement. The interface supports the following operations:

```
uint32_t ll_read(ll_driver_t *drv,
                 uint8_t *data,
                 uint32_t data_size);
uint32_t ll_write(ll_driver_t *drv,
                  uint8_t *data,
                  uint32_t data_size);
uint32_t ll_data_readable(ll_driver_t *drv);
uint32_t ll_data_writeable(ll_driver_t *drv);
bool ll_is_synchronous(ll_driver_t *drv);
```

For a synchronous driver such as the LED, `ll_read` and `ll_write` can be called at any time to read or write data from the driver. Likewise, the `ll_data_readable` and `ll_data_writeable` always returns a value greater than zero in the synchronous case. The `ll_is_synchronous` function is used by the SenseVM runtime to distinguish asynchronous, interrupt-driven drivers from the synchronous ones and handle them accordingly.

The asynchronous case is more interesting. Say that the program wants to write a value to an asynchronous driver, for example a UART (serial communication). The low-level UART implementation may be using a buffer that can be full and `ll_data_writeable` will in this case return zero. The SenseVM task that tried to write data will now block. Tasks that are blocked on either asynchronous read or write have to be woken up when the lower-level driver implementation has *produced* or *is ready to receive* data. These state changes are usually signaled using interrupts.

The interrupt service routines (ISRs) associated with asynchronous drivers talk to the SenseVM RTS using a message queue and a driver message type called `ll_driver_msg_t` akin to the Medusa system [3]. A simple event such as "a button has been pressed" can be fully represented in the message, but if the message signals that a larger buffer is filled with data that needs processing, this data can be accessed through the `ll_read/write` interface.

The SenseVM scheduler runs within a Zephyr thread. This thread is the owner of the message queue to which all the drivers are sending messages. When the Zephyr thread starts the SenseVM scheduler, it passes along a pointer to a `read_message` function that reads a message from the message queue or blocks if the queue is empty. Thus, all code that relies on Zephyr is confined to the low-level drivers and to the thread that runs the scheduler.

4 Evaluation

Evaluations are run on the STM32F4 microcontroller with a Cortex M4 core at 168MHz (STM32F407G-DISC1) and an nRF52840 microcontroller with an 80MHz Cortex M4 core (UBLOX BMD340). We should point out that the button-blinky program runs *portably* on both of these boards with no change in the code required.

4.1 Power Consumption

Figure 3 shows the power consumption of the button-blinky program measured in the nRF52 board for (i) a polling implementation of the program written in C, (ii) the interrupt-based implementation written in C [7] and (iii) the SenseVM implementation. The measurements were obtained using a Ruideng UM25C ammeter where the values provided are momentary and read after they stabilised.

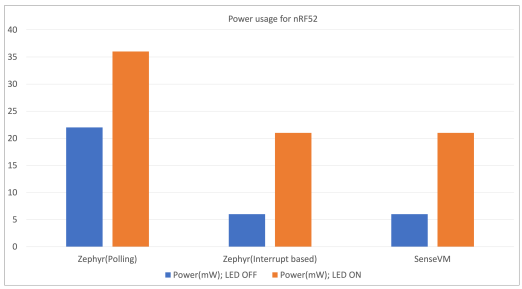


Figure 3. Power consumption with LED OFF and ON in mW

The figure shows that the polling implementation consumes up to four times more power when the LED is switched OFF and up to twice as much power when the LED is ON compared to the SenseVM implementation. The C interrupt based implementation and the SenseVM program have identical power usage. However, the C program is 127 lines of callback-based code compared to the 13 line SenseVM program shown in Section 2.3.

4.2 Response Time

Table 1 shows two different implementations of button-blinky written directly in C and Zephyr in comparison to the SenseVM implementation. The values in the table represent the time it takes from a button being pressed until the associated LED turns on. The values were obtained using a UNI-T UTD2102CEX (100MHz, 1GS/s oscilloscope) that has not been calibrated other than the original factory calibrations. We elaborate on the oscilloscope measurements in Appendix A

Table 1. Response time for button-blinky program in μsecs

	Zephyr	Zephyr (Interrupt based)	SenseVM
STM32F4	0.88	9.5	37.7
NRF52	1.9	17.3	61.6

The response time for the polling based implementation is considerably faster than the rest. This comes at the cost of being much more power intensive (upto 4 times for this program). The SenseVM response time is up to 4 times slower than the more realistic interrupt based C program. This slowdown is majorly attributed to two factors i) the interpretation overhead and ii) the *stop-the-world* garbage collector. As future work, we plan to experiment with various *ahead-of-time* and *just-in-time* compilation strategies to speed up the bytecode interpretation and thus improve the response times. We also hope that more incremental, real-time garbage collectors can further speed up the response times.

4.3 Memory Usage

SenseVM statically allocates the stack, heap, number of channels, number of threads etc and allows configuration of their sizes. As a result, when comparing memory usage between SenseVM and Zephyr, a lot of the memory reported is not currently being used but statically allocated for *potential use* by the VM. We show some memory usage statistics in Table 2.

Table 2. Memory usage of button-blinky in KiB (= 1024 bytes)

	Flash	RAM
SenseVM	37.4	27.6
Zephyr message queue	16.5	4.55

While obtaining the data in Table 2, SenseVM was configured with a heap of 1024 bytes and 1024 bytes for use by the stacks of each context. Additionally we set aside 9600 bytes for channels, of which, only 2 channels (192 bytes) are used and the remaining 9408 bytes remain unused. The button-blinky program uses 1 thread, 2 channels and interacts with 2 drivers (LED and button), while in the VM we have statically allocated space for 4 threads, 100 channels and metadata for 16 drivers. This additional space allocation is done keeping in mind multi-threaded programs with larger inter-process networks communicating via several channels using various drivers.

5 Related Work

Among statically-typed languages, Varoumas et.al [30] presented a virtual machine that can host the OCaml language on PIC microcontrollers. They have extended OCaml with a deterministic model of concurrency - OCaLustre [29]. This line of work, however, does not mitigate the pain associated with *callback hell*, as the concurrency model does not extend to the interrupts and their handlers. In SenseVM, we unify the notion of concurrency and I/O (callback-based and otherwise) via message-passing to simplify callback-oriented programming, prevalent in microcontrollers.

In the dynamically-typed language world there exists Medusa [3], for programming the TI Stellaris microcontrollers, which is much closer to our line of work by unifying concurrency and I/O. The difference between Medusa and SenseVM boils down to the comparison between actor based,

asynchronous message passing systems and the synchronous message passing model, which has been discussed in Section 3.2.3. Moreover, the static typing of our frontend language enables *typed channels* to perform static checks on message contents, done at runtime in Medusa. One should note, however, that actor based systems excel at distributed computing - a more failure-prone and harder form of concurrent computing.

Programming environments like VeloxVM [28] focus on the safety and security of microcontroller programming . We leave our investigations on the formal safety and security aspects of microcontroller programming as future work. There also exists work to improve the portability, debuggability and live-code updating capability of microcontrollers using WebAssembly [25].

Our previous work [22] has attempted to use the Functional Reactive Programming (FRP) paradigm [6] for the programming of microcontrollers. However, it suffered from the performance penalties of *polling*, owing to the pull-based semantics of FRP, which has been addressed in this higher-order concurrency model.

The higher-order concurrency model, first introduced by Reppy [20], has been primarily used for programming GUIs such as eXene [8]. It has also found application in more experimental distributed implementations of SML such as DML [4]. We provide, to the best of our knowledge, the first implementation of the model for natively programming microcontrollers. In all previous implementations of higher-order concurrency, external I/O has been represented using stream-based I/O primitives on top of SML's standard I/O API [21]. We differ in this aspect by modelling any external I/O device as a process in itself (connected using `spawnExternal`) and communicating via the standard message passing interface, making the model more uniform.

Regarding portability, there exist JVM implementations like Jamaica VM [24] for running portable Java code across various classes of embedded systems. The WebAssembly project has also spawned sub-projects like the WASM micro-runtime [2] to allow languages that compile to WebAssembly to run portably on microcontrollers. It should be pointed out that while general-purpose languages like Javascript can execute on ARM architectures by compiling to WebAssembly, it is still the case that neither the language nor the VM offers any intrinsic model of concurrency unified with I/O, analogous to the one provided by SenseVM.

6 Conclusion

We have presented SenseVM, a bytecode-interpreted virtual machine to simplify the concurrent and reactive programming of microcontrollers. We introduce the higher-order concurrency model to the world of microcontroller programming and show the feasibility of implementing the model by presenting the power usage, response time measurements and memory footprint of the VM on an interrupt-driven, callback based program. We additionally present our VM to address the portability concerns that plague low-level C and assembly microcontroller programs. We demonstrate the portability of our VM by running the same program, unchanged, on the `nRF52840` and `STM32F4` microcontrollers. In future work, we hope to experiment with real-time programming on microcontrollers by exploring more deterministic approaches to memory management like regions and real-time garbage collectors.

Acknowledgments

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Chalmers Gender Initiative for Excellence (Genie).

A Response Time Measurement Data

In the pictures below, the blue line represents the button and the yellow line represents the LED. The measurement is taken from the rising edge of the button line, to the rising edge of the LED while trying to (when possible) cut the slope of the edge in the middle. All pictures are captured on the same UNI-T UTD2102CEX (100MHz, 1GS/s oscilloscope).

The oscilloscope used here is not pleasant to gather large datasets. To the best of our knowledge it does not support a network connection and API access.

The occurrence of outliers in our measurements is discussed in the next section. A series of 25 SenseVM button-blinky response time measurements, filtered from outliers, were collected on the STM32F4 microcontroller. Out of these 25 measurements 23 were measured at 37.7μs and the other two came in at 37.4μs and 37.5μs.

B Outliers

While repeatedly running and capturing the response time of the button-blinky program on the STM32F4, periodical outliers occur. In a sequence of 108 experiments, 11 showed

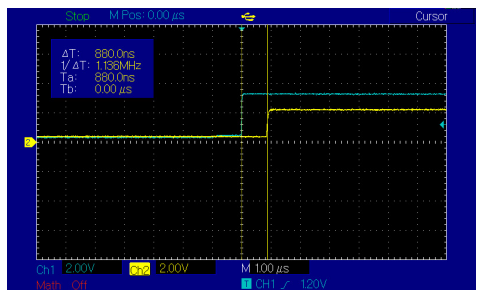


Figure 4. STM32F4 Discovery: Zephyr implementation of button-blinky that constantly polls button and updates LED.

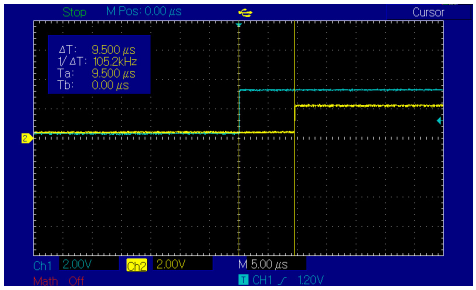


Figure 5. STM32F4 Discovery: Zephyr implementation of button-blinky using interrupt and message queue.

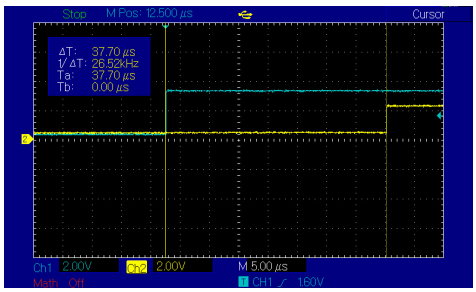


Figure 6. STM32F4 Discovery: SenseVM implementation of button-blinky.

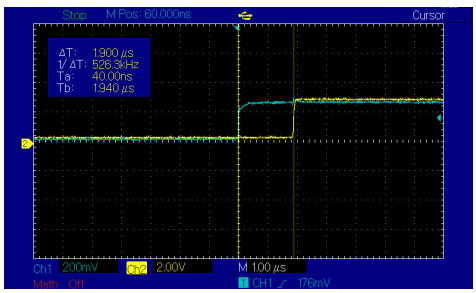


Figure 7. NRF52: Zephyr implementation of button-blinky that constantly polls button and updates LED.

an outlier. The occurrence of these anomalies is likely to be caused by the garbage collector performing a mark phase. There is also a chance that if a large part of the heap is in use, the allocation operation becomes expensive as it may need to search a larger space before finding a free cell using the lazy sweep algorithm.

The figure below shows a response time of 96μs compared to the approximately 38μs response time found in most cases.

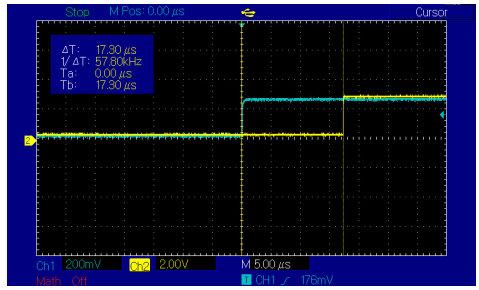


Figure 8. NRF52: Zephyr implementation of button-blinky using interrupt and message queue.

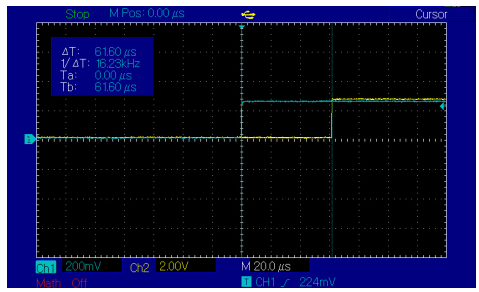


Figure 9. NRF52: SenseVM version of button-blinky.

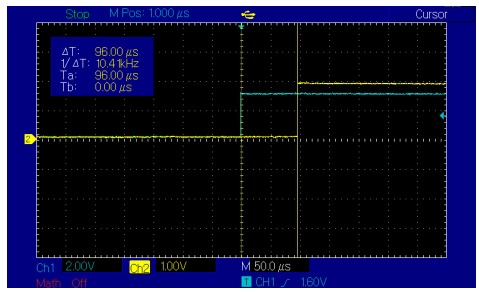


Figure 10. STM32F4 Discovery: Outlier measured while running SenseVM implementation of button-blinky.

The worst outlier found so far was measured to 106.8μs but these seem to be rare and most outliers fell somewhat short of 96μs. As future work better measurement methodology will be developed and applied. Preferably such a setup should allow scripting of a large number of automated tests.

References

[1] 2016. *Zephyr OS.* https://www.zephyrproject.org/

[2] 2019. *WAMR - WebAssembly Micro Runtime.* https://github.com/bytecodealliance/wasm-micro-runtime

[3] Thomas W Barr and Scott Rixner. 2014. Medusa: Managing concurrency and communication in embedded systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 439–450.

[4] Robert Cooper and Clifford Krumvieda. 1992. Distributed programming with asynchronous ordered channels in distributed ML. In *ACM SIGPLAN Workshop on ML and Its Applications*. 134–150.

[5] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. 1985. The Categorical Abstract Machine. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 50–64. https://doi.org/10.1007/3-540-15975-4_29

[6] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 263–273. https://doi.org/10.1145/258948.258973

[7] Zephyr examples. 2021. *Zephyr Button Blinky.* https://gist.github.com/Abhiroop/d83755d7a5703f704fbfb9c3d116d87c

[8] Emden R Gansner and John H Reppy. 1991. eXene. In *Third International Workshop on Standard ML, Pittsburgh, PA*.

[9] Damien George. 2014. *Micropython.* https://micropython.org/

[10] Carl Hewitt. 2010. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459* (2010).

[11] Ralf Hinze. 1993. *The Categorical Abstract Machine: Basics and Enhancements.* Technical Report. University of Bonn.

[12] Charles Antony Richard Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

[13] R John M Hughes. 1982. A semi-incremental garbage collection algorithm. *Software: Practice and Experience* 12, 11 (1982), 1081–1082.

[14] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

[15] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[16] Xavier Leroy. 1997. The Caml Light system release 0.74. *URL: http://caml. inria. fr* (1997).

[17] Henry Lieberman and Carl Hewitt. 1983. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (1983), 419–429. https://doi.org/10.1145/358141.358147

[18] Tommi Mikkonen and Antero Taivalsaari. 2008. Web Applications - Spaghetti Code for the 21st Century. In *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye (Eds.). IEEE Computer Society, 319–328. https://doi.org/10.1109/SERA.2008.16

[19] Norman Ramsey. 1990. *Concurrent programming in ML.* Technical Report. Princeton University.

[20] John H Reppy. 1992. *Higher-order concurrency.* Technical Report. Cornell University.

[21] John H Reppy. 1993. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer, 165–198.

[22] Abhiroop Sarkar and Mary Sheeran. 2020. 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, 12:1–12:16. https://doi.org/10.1145/3414080.3414092

[23] Herbert Schorr and William M. Waite. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM* 10, 8 (1967), 501–506. https://doi.org/10.1145/363534.363554

[24] Fridtjof Siebert. 1999. Hard Real-Time Garbage-Collection in the Jamaica Virtual Machine. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*. IEEE Computer Society, 96–102. https://doi.org/10.1109/RTCSA.1999.811198

[25] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, Antony L. Hosking and Irene Finocchi (Eds.). ACM, 27–36. https://doi.org/10.1145/3357390.3361029

[26] VDC Research Survey. 2011. *Embedded Engineer Survey Results.* https://blog.vdcresearch.com/embedded_sw/2011/06/2011-embedded-engineer-survey-results-programming-languages-used-to-develop-software.html

[27] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[28] Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. *J. Netw. Comput. Appl.* 118 (2018), 61–73. https://doi.org/10.1016/j.jnca.2018.06.001

[29] Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. 2016. Concurrent Programming of Microcontrollers, a Virtual Machine Approach. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. 711–720.

[30] Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. 2018. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*.

[31] Gordon Williams. 2012. *Espruino.* http://www.espruino.com/

[32] Zephyr. 2016. *Zephyr Mutex API.* https://docs.zephyrproject.org/apidoc/latest/group__mutex__apis.html

[33] Zephyr. 2016. *Zephyr Semaphore API.* https://docs.zephyrproject.org/apidoc/latest/group__semaphore__apis.html

[34] Zephyr. 2016. *Zephyr Semaphore API.* https://docs.zephyrproject.org/apidoc/latest/group__thread__apis.html

Chapter 4

Synchron - An API and Runtime for Embedded Systems

Synchron - An API and Runtime for Embedded Systems

Abhiroop Sarkar
Chalmers University, Sweden
sarkara@chalmers.se

Bo Joel Svensson
Chalmers University, Sweden
joels@chalmers.se

Mary Sheeran
Chalmers University, Sweden
mary.sheeran@chalmers.se

Abstract

Programming embedded systems applications involves writing concurrent, event-driven and timing-aware programs. Traditionally, such programs are written in low-level machine-oriented programming languages like C or Assembler. We present an alternative by introducing Synchron, an API that offers high-level abstractions to the programmer while supporting the low-level infrastructure in an associated runtime system and one-time-effort drivers.

Embedded systems applications exhibit the general characteristics of being (i) concurrent, (ii) I/O–bound and (iii) timing-aware. To address each of these concerns the Synchron API consists of three components - (1) a Concurrent ML (CML) inspired message-passing concurrency model, (2) a message-passing–based I/O interface that translates between low-level interrupt based and memory-mapped peripherals and (3) a timing operator `syncT`, that marries CML's `sync` operator with timing windows inspired from the TinyTimber kernel.

We implement the Synchron API as the bytecode instructions of a virtual machine called SynchronVM. SynchronVM hosts a Caml-inspired functional language as its frontend language, and the backend of the VM supports the STM32F4 and NRF52 microcontrollers, with RAM in the order of hundreds of kilobytes. We illustrate the expressiveness of the Synchron API by showing examples of expressing state machines commonly found in embedded systems. The timing functionality is demonstrated through a music programming exercise. Finally, we provide benchmarks on the response time, jitter rates and power usage of the SynchronVM.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Runtime environments; Computer systems organization → Real-time languages; Software and its engineering → Concurrent programming languages

Keywords and phrases real-time, concurrency, functional programming, runtime, virtual machine

Digital Object Identifier 10.4230/LIPIcs.OOOOO.2022.23

Funding This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Chalmers Gender Initiative for Excellence (Genie).

1 Introduction

Embedded systems are ubiquitous and are pervasively found in application areas like IoT, industrial machinery, cars, robotics, etc. Applications running on embedded systems usually embody three common characteristics:

1. They are *concurrent* in nature.
2. They are predominantly *I/O–bound* applications.
3. A large subset of such applications are *timing-aware*.

This list is by no means exhaustive but captures a prevalent theme among embedded applications. Programming these applications involves interaction with callback-based driver APIs like the following from the Zephyr RTOS[11]:

Listing 1 A callback-based GPIO driver API

```
int gpio_pin_interrupt_configure(const struct device *port
                                 , gpio_pin_t pin
                                 , gpio_flags_t flags);
void gpio_init_callback(struct gpio_callback *callback
                        , gpio_callback_handler_t handler
                        , gpio_port_pins_t pin_mask);
int gpio_add_callback(const struct device *port
                      , struct gpio_callback *callback);
```

Programming with such APIs in low-level languages like C leads to complicated state machines that, even for relatively small programs, result in difficult-to-maintain and complex state-transition tables. Moreover, C programmers use error-prone shared-memory primitives like *semaphores* and *locks* to mediate interactions that occur between the callback-based driver handlers.

More modern microcontroller runtimes like MicroPython [12] and Espruino (Javascript) [35] support higher-order functions and handle callback-based APIs in the following way:

Listing 2 Driver interactions using Micropython

```
1 def callback(x):
2 #...callback body with nested callbacks...
3
4 extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING
5                     , pyb.Pin.PULL_UP, callback)
6 ExtInt.enable()
```

The function `callback(x)` from Line 1 can in turn define a callback action `callback2`, which can further define other callbacks leading to a cascade of nested callbacks. This leads to a form of *accidental* complexity, colloquially termed as *callback-hell* [20].

We present Synchron, an API that attempts to address the concerns about callback-hell and shared-memory communication while targeting the three characteristics of embedded programs mentioned earlier by a combination of:

1. A message-passing–based concurrency model inspired from Concurrent ML.
2. A message-passing–based I/O interface that unifies concurrency and I/O.
3. A notion of time that fits the message-passing concurrency model.

Concurrent ML (CML) [25] builds upon the synchronous message-passing–based concurrency model CSP [15] but adds the feature of composable first-class *events*. These first-class events allow the programmer to tailor new concurrency abstractions and express application-specific protocols. Moreover, a synchronous concurrency model renders linear control-flow to a program, as opposed to bottom-up, non-linear control flow exhibited by asynchronous callback-based APIs.

Synchron extends CML's message-passing API for software processes to I/O and hardware interactions by modelling the external world as a process through the `spawnExternal` operator. As a result, the standard message-passing functions such as `send, receive` etc. are applicable for handling I/O interactions such as asynchronous driver interrupts. The overall API design allows efficient scheduling and limited power usage of programs via an associated runtime.

For timing, Synchron has the `syncT` operator, drawing inspiration from the TinyTimber kernel [19] that allows the specification of baseline and deadline windows for invocation of a method in a class. In TinyTimber, `WITHIN( B, D, &obj, meth, 123 );` expresses the

desire that method `meth` should be run at the earliest at time `B` and finish within a duration of `D`. Our adaptation of this API, `syncT B D evt`, takes a baseline, deadline and a CML *event* (evt) as arguments and obeys similar semantics as `WITHIN`.

The Synchron API is implemented in the form of a bytecode-interpreted virtual machine (VM) called SynchronVM. The bytecode instructions of the VM correspond to the various operations of the Synchron API, such that any language hosted on the VM can access Synchron's concurrency, I/O and timing API for embedded systems.

Internally, the SynchronVM runtime manages the scheduling and timing of the various processes, interrupt handling, memory management, and other bookkeeping infrastructure. Notably, the runtime system features a low-level bridge interface that abstracts it from the platform-dependent specifics. The bridge translates low-level hardware interrupts or memory-mapped I/O into software messages, enabling the SynchronVM application process to use the message-passing API for low-level I/O.

Contributions

- We identify three characteristic behaviours of embedded applications, namely being (i) concurrent, (ii) I/O–bound, and (iii) timing-aware, and propose a combination of abstractions (the Synchron API) that mesh well with each other and address these requirements. We introduce the API in Section 3.
- **Message-passing–based I/O.** We present a uniform message-passing framework that combines *concurrency* and *callback–based I/O* to a single interface. A software message or a hardware interrupt is identical in our programming interface, providing the programmer with a simpler message-based framework to express concurrent hardware interactions. We show the I/O API in Section 3.2 and describe the core runtime algorithms to support this API in Section 4.
- **Declarative state machines for embedded systems.** Combining CML primitives with our I/O interface allows presenting a declarative framework to express state machines, commonly found in embedded systems. We illustrate examples of representing finite-state machines using the Synchron API in Sections 6.1 and 6.2.
- **Evaluation.** We implement the Synchron API and its associated runtime within a virtual machine, SynchronVM, described in Section 5. We illustrate the practicality and expressivity of our API by presenting three case studies in Section 6, which runs on the STM32 and NRF52 microcontroller boards. Finally, we show response-time, power usage, jitter-rates, and load testing benchmarks on the SynchronVM in Section 7

2 Motivation

• **Concurrency and IO.** In embedded systems, concurrency takes the form of a combination of callback handlers, interrupt service routines and possibly a threading system, for example threads as provided by ZephyrOS, ChibiOS or FreeRTOS. The callback style of programming is complicated but offers benefits when it comes to energy efficiency. Registering a callback with an Interrupt Service Routine (ISR) allows the processor to go to sleep and conserve power until the interrupt arrives.

An alternate pattern to restore the linear control flow of a program is the event-loop pattern. As the name implies, an event-loop based program involves an infinitely running loop that handles interrupts and dispatches the corresponding interrupt-handlers. An event-loop based program involves some delicate plumbing that connects its various components. Listing 3 shows a tiny snippet of the general pattern.

Listing 3 Event Loop

```
void eventLoop(){
while (1) {
 switch(GetNextEvent()) {
    case GPIO1 : GPIO1Handler();
    case GPIO2 : GPIO2Handler();
    ....
    default : goToSleep(); // no events
}}}

GPIO1Handler(){ ... } // must not block
GPIO2Handler(){ ... } // must not block

//when interrupt arrives write to event queue
GPIO1_IRQ(){....}
GPIO2_IRQ(){....}
```

Programs like the above are an improvement over callbacks, as they restore the linear control-flow of a program, which eases reasoning. However, such programs have a number of weaknesses - (i) they enforce constraints on the blocking and non-blocking behaviours of the event handlers, (ii) programmers have to hand-code elaborate plumbings between the interrupt-handlers and the event-queue, (iii) they are highly *inextensible* as extension requires code modifications on all components of the event-loop infrastructure, and (iv) they are instances of clearly concurrent programs that are written in this style due to lack of native concurrency support in the language.

Although there are extensions of C to aid the concurrent behaviour of event-loops, such as protothreads [8] or `FreeRTOS Tasks`, the first three listed problems still persist. The main infinite event loop unnecessarily induces a tight coupling between unrelated code fragments like the two separate handlers for GPIO1 and GPIO2. Additionally, this pattern breaks down the abstraction boundaries between the handlers.

• **Time.** A close relative of concurrent programming for embedded systems is *real-time programming*. Embedded systems applications such as digital sound cards routinely exhibit behaviour where the time of completion of an operation determines the correctness of the program. Real-time programs, while concurrent, differ from standard concurrent programs by allowing the programmer to override the fairness of a *fair* scheduler.

For instance, the `FreeRTOS Task` API allows a programmer to define a static *priority* number, which can override the *fairness* of a task scheduler and customise the emergency of execution of each thread. However, with a limited set of priorities numbers (1 - 5) it is very likely for several concurrent tasks to end up with the same priority, leading the scheduler to order them fairly once again. A common risk with priority-based systems is to run into the *priority inversion problem* [31], which can have fatal consequences on hard real-time scenarios. On the other hand, high-level language platforms for embedded systems such as MicroPython [12] do not provide any language support for timing-aware computations.

Problem Statement. We believe there exists a gap for a high-level language that can express concurrent, I/O–bound, and timing-aware programs for programming resource-constrained embedded systems. We outline our key idea to address this gap below.

2.1 Key Ideas

Our key idea is the Synchron API, which adopts a synchronous message-passing concurrency model and extends the message-passing functionality to all I/O interactions. Synchron also introduces *baselines* and *deadlines* for the message-passing, which consequently introduces a

notion of time into the API. The resultant API is a collection of nine operations that can express (i) concurrency, (ii) I/O, and (iii) timing in a uniform and declarative manner.

The external world as processes. The Synchron API models all external drivers as processes that can communicate with the software layer through message-passing. Synchron's `spawnExternal` operator treats an I/O peripheral as a process and a hardware interrupt as a message from the corresponding process. Fig. 1 illustrates the broad idea.

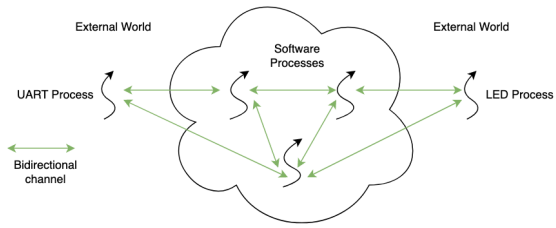


Figure 1 Software processes and hardware processes interacting

The above design relieves the programmer from writing complex callback handlers to deal with asynchronous interrupts. The synchronous message-passing–based I/O further renders a linear control-flow to I/O-bound embedded-system programs, allowing the modelling of state machines in a declarative manner. Additionally, the message-passing framework simplifies the hazards of concurrent programming with shared-memory primitives (like `FreeRTOS semaphores`) and the associated perils of maintaining intricate locking protocols.

Hardware-Software Bridge. The Synchron runtime enables the seamless translation between software messages and hardware interrupts. The runtime does hardware interactions through a low-level software *bridge* interface, which is implemented atop the drivers supplied by an OS like Zephyr/ChibiOS. The *bridge* layer serialises all hardware interrupts into the format of a software message, thereby providing a uniform message-passing interaction style for both software and hardware messages. Fig. 2 shows the overall architecture of Synchron.

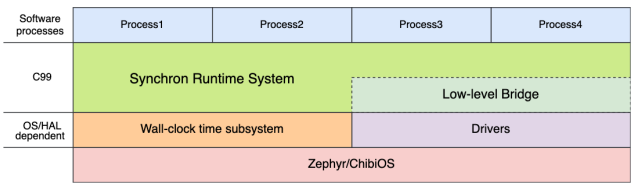


Figure 2 The Synchron Architecture

Timing. The final key component of the Synchron API is the real-time function, `syncT`, that instead of using a static priority for a thread (like Ada, RT-Java, FreeRTOS, etc.), borrows the concept of a dynamic priority specification from TinyTimber [19].

The `syncT` function allows specifying a *timing window* by stating the baseline and deadline of message communication between processes. The logical computational model of Synchron assumes to take zero time and hence the time required for communication determines the timing window of execution of the entire process. As the deadline of a process draws near, the Synchron runtime can choose to dynamically change the priority of a process while it is running. Fig. 3 illustrates the idea of dynamic priority-change.

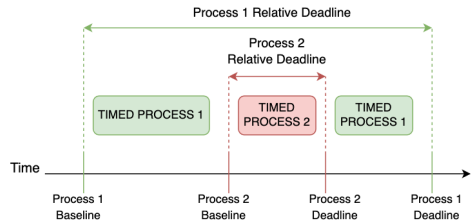


Figure 3 Dynamic priority change with `syncT`

Fig. 3 above shows how a scheduler can choose to prioritise a second process over a running, *timed* process, even though the running process has a deadline in the future. In practice, a scheduler needs to be able to pause and resume processes to support the above, which is possible in the Synchron runtime. The `syncT` function, thus, fits fairly well with the rest of the API and provides a notion of time to the overall programming interface.

The combination of `syncT`, `spawnExternal` and the CML-inspired synchronous message-passing concurrency model constitutes the Synchron API that allows declarative specification of embedded applications. We suggest that this API is an improvement, in terms of expressivity, over the currently existing languages and libraries on embedded systems and provide illustrative examples to support this in Section 6. We also provide benchmarks on the Synchron runtime in Section 7. Next, we discuss the Synchron API in more detail.

3 The Synchron API

3.1 Synchronous Message-Passing and Events

We begin by looking at a standard synchronous message-passing API, like Hoare's CSP [15] -

```
spawn   : (() -> ()) -> ThreadId
channel : () -> Channel a
sendMsg : Channel a -> a -> ()
recvMsg : Channel a -> a
```

In the above type signatures the parameter, `a` indicates a polymorphic type. The call to `spawn` allows the creation of a new process whose body is represented by the $(() \to ())$ type. The `channel ()` call creates a blocking *channel* along which a process can send or receive messages using `sendMsg` and `recvMsg` respectively. A channel blocks until a sender has a corresponding receiver and vice-versa. Multi-party communication in CSP is enabled using the *nondeterministic choice* operation that races between two sets of synchronous operations and chooses the one that succeeds first.

However, there is a fundamental conflict between procedural abstraction and the *choice* operation. Using a procedure to represent a complex protocol involving multiple *sends* and *receives* hides the critical operations over which a *choice* operation is run. On the other hand, exposing the individual *sends* and *receives* prevents the construction of protocols with strict message-ordering constraints (see Appendix **??**). Reppy discusses this issue [25] in depth and provides a solution in Concurrent ML (CML), which is adopted by the Synchron API.

The central idea is to break the act of synchronous communication into two steps:

(i) Expressing the intent of communication as an *event*-value
(ii) Synchronising between the sender and receiver via the *event*-value

The first step above results in the creation of a type of value called an `Event`. An *event* is a first-class value in the language akin to the treatment of higher-order functions in functional languages. Reppy describes events as "first-class synchronous operations" [25]. Adapting this idea, the type signature of message sending and receiving in the Synchron API becomes :

send : Channel a → a → Event ()
recv : Channel a → Event a

Given a value of type `Event`, the second step of synchronising between processes and the consequent act of communication is accomplished via the `sync` operation, whose type is :

sync : Event a → a

Intuitively, an equivalence can be drawn between the message passing in CSP and the CML-style message passing (as adopted in the Synchron API) using function composition:

```
sync . (send c) ≡ sendMsg c
sync . recv     ≡ recvMsg
```

The advantage of representing communication as a first-class value is that event-based combinators can be used to build more elaborate communication protocols. In the same vein as higher-order functions like *function composition (.)* and *map*, events support composition via the following operators in the Synchron API :

choose : Event a → Event a → Event a
wrap : Event a → (a → b) → Event b

The `choose` operator is equivalent to the *choice* operation in CSP and the `wrap` operator can be used to apply a post-synchronisation operation (of type $a \rightarrow b$). A large tree of *events* representing a communication protocol can be built in this framework as follows :

```
protocol : Event ()
protocol =
  choose (send c1 msg1)
         (wrap (recv c2) (λ msg2 -> sync (send c3 msg2)))
```

Using events, the above protocol involving multiple *sends* and *receives* was expressible as a procedural abstraction while still having the return type of `Event ()`. A consumer of the above protocol can further use the nondeterministic choice operator, `choose`, and *choose* among multiple protocols. This combination of a composable functional programming style and CSP-inspired multiprocess program design allows this API to represent callback-based, state machine oriented programs in a declarative manner.

Comparisons between Events and Futures. The fundamental difference between events and futures is that of *deferred communication* and *deferred computation* respectively. A future aids in asynchronous computations by encapsulating a computation whose value is made available at a future time. On the other hand, an event represents deferred communication as a first-class entity in a language. Using the `wrap` combinator, it is possible to chain lambda functions capturing computations that should happen post-communication as well. However, events are fundamentally building blocks for communication protocols.

3.2 Input and Output

The I/O component of Synchron is deeply connected to the Synchron runtime. Hence, we mention parts of the low-level runtime implementation while describing the I/O interface.

In the Synchron API, I/O is expressed using the same events as are used for inter-process communication. Each I/O device is connected to the running program using a primitive we call `spawnExternal` as a hint that the programmer can think of, for example, an LED as a process that can receive messages along a channel. Each *external* process denotes an underlying I/O device that is limited to send and receive messages along one channel.

The `spawnExternal` primitive takes the channel to use for communication with software and a driver and returns a "ThreadId" for symmetry with `spawn`.

spawnExternal : Channel a → Driver → ExternalThreadId

The first parameter supplied to `spawnExternal` is a designated fixed channel along which the external process shall communicate. The second argument requires some form of identifier to uniquely identify the driver. This identifier for a driver tends to be architecture-dependent. For instance, when using low-level memory-mapped I/O, reads or writes to a memory address are used to communicate with a peripheral. So the unique memory address would be an identifier in that case. On the other hand, certain real-time operating system (such as `FreeRTOS` or `Zephyr`) can provide more high-level abstractions over a memory address. In the Synchron runtime, we number each peripheral in a monotonically increasing order, starting from 0. So our `spawnExternal` API becomes:

```
type DriverNo = Int
spawnExternal : Channel a -> DriverNo -> ExternalThreadId
```

In the rest of the paper, we will use suggestive names for drivers like `led0`, `uart1`, etc instead of plain integers for clarity. We have ongoing work to parse a file describing the target board/MCU system, automatically number the peripherals and emit typed declaration like $led0 = LED\ 0$ that can be used in the `spawnExternal` API.

To demonstrate the I/O API for asynchronous drivers, we present a standard example of the *button-blinky* program. The program matches a button state to an LED so that when the button is down, the LED is on, otherwise the LED is off:

Listing 4 Button-Blinky using the Synchron API

```
1  butchan = channel ()
2  ledchan = channel ()
3
4  glowled i = sync (send ledchan i)
5
6  f : ()
7  f = let _ = sync (wrap (recv butchan) glowled) in f
8
9  main =
10   let _ = spawnExternal butchan 0 in
11   let _ = spawnExternal ledchan 1 in f
```

Listing 4 above spawns two hardware processes - an LED process and a button process. It then calls the function `f` which arrives at line 8 and waits for a button press. During the waiting period, the scheduler can put the process to sleep to save power. When the button interrupt arrives, the Synchron runtime converts the hardware interrupt to a software message and wakes up process `f`. It then calls the `glowled` function on line 4 that sends a switch-on message to the LED process and recursively calls `f` infinitely.

The above program represents an asynchronous, callback-based application in an entirely synchronous framework. The same application written in C, on top of the Zephyr OS, is more than 100 lines of callback-based code [10]. A notable aspect of the above program is the lack of any non-linear callback-handling mechanism.

The structure of this program resembles the event-loop pattern presented in Listing 3 but fixes all of its associated deficiencies - (1) all Synchron I/O operations are blocking; the runtime manages their optimal scheduling, not the programmer, (2) the internal plumbing related to interrupt-handling and queues are invisible to the programmer, (3) the program is highly extensible; adding a new interrupt handler is as simple as defining a new function.

3.3 Programming with Time

Real-time languages and frameworks generally provide a mechanism to override the fairness of a *fair* scheduler. A typical fair scheduler abstracts away the details of prioritising processes.

However, in a real-time scenario, a programmer wants to precisely control the response-time of certain operations. So the natural intuition for real-time C-extensions like FreeRTOS *Tasks* or languages like Ada is to delegate the scheduling control to the programmer by allowing them to attach a priority level to each process.

The priority levels involved decides the order in which a tie is broken by the scheduler. However, with a small fixed number of priority levels it is likely for several processes to end up with the same priority, leading the scheduler to order them fairly again within each level.

Another complication that crops up in the context of priorities is the *priority inversion problem* [31]. Priority inversion is a form of resource contention where a high-priority thread gets blocked on a resource held by a low-priority thread, thus allowing a medium priority thread to take advantage of the situation and get scheduled first. The outcome of this scenario is that the high-priority thread gets to run after the medium-priority thread, leading to possible program failures.

The Synchron API admits the *dynamic* prioritisation of processes, drawing inspiration from the TinyTimber kernel [21]. TinyTimber allows specifying a *timing window* expressed as a baseline and deadline time, and a scheduler can use this timing window to determine the runtime priority of a process. The timing window expresses the programmer's wish that the operation is started at the *earliest* on the baseline and *no later* than the deadline.

In Synchron, a programmer specifies a *timing window* (of the wall-clock time) during which they want message synchronisation, that is the rendezvous between message sender and receiver, to happen. We do this with the help of the timed-synchronisation operator, `syncT`, with type signature:

syncT : Time → Time → Event a → a

Comparing the type signature of `syncT` with that of `sync` :

```
1 syncT : Time -> Time -> Event a -> a
2 sync  :                 Event a -> a
```

The two extra arguments to `syncT` specify a lower and upper bound on the *time of synchronisation* of an event. The two arguments to `syncT`, of type `Time`, express the relative times calculated from the current wall-clock time. The first argument represents the *relative baseline* - the earliest time instant from which the event synchronisation should begin. The second argument specifies the *relative deadline*, i.e. the latest time instant (starting from the baseline), by which the synchronisation should start. For instance,

```
syncT (msec 50) (msec 20) timed_ev
```

means that the event, *timed_ev*, should begin synchronisation at the earliest 50 milliseconds and the latest $50 + 20$ milliseconds from *now*. The *now* concept is based on a thread's local view of what time it is. This thread-local time (T_{local}) is always less than or equal to wall-clock time ($T_{absolute}$). When a thread is spawned, its thread-local time, T_{local}, is set to the wall-clock time, $T_{absolute}$.

While a thread is running, its local time is frozen and unchanged until the thread executes a timed synchronisation, a `syncT` operation where time progresses to $T_{local} + baseline$.

```
process1 _ =
  let _ = s1 in -- T_local = 0
  let _ = s2 in -- T_local = 0
  let _ = syncT (msec 50) (usec 10) ev1 in
  process1 () -- T_local = 50 msec
```

The above illustrates that the *untimed* operations `s1` and `s2` have no impact on a thread's view of what time it is. In essence, these operations are considered to take no time, which is a reference to *logical* time and not the physical time. Synchron shares this logical computational model with other systems such as the synchronous languages [7] and ChucK [33].

In practice, this assumption helps control jitter in the timing as long as the timing windows specified on the synchronisation is large enough to contain the execution time of `s1`, `s2`, the synchronisation step and the recursive call. Local and absolute time must meet up at key points for this approach to work. Without the two notions of time meeting, local time would drift away from absolute time in an unbounded fashion. For a practical implementation of `syncT`, a scheduler needs to meet the following requirements:

- The scheduler should provide a mechanism for overriding fair scheduling.
- The scheduler must have access to a wall-clock time source.
- A scheduler should attempt to schedule synchronisation such that local time meets up with absolute time at that instant.

We shall revisit these requirements in Section 5 when describing the scheduler within the Synchron runtime. Next, we shall look at a simple example use of `syncT`.

Blinky

The well-known *blinky* example, shown in Listing 5, involves blinking an LED on and off at a certain frequency. Here we blink every one second.

Listing 5 Blinky with `syncT`

```
not 1 = 0
not 0 = 1

ledchan = channel ()

sec  n = n * 1000000
usec n = n -- the unit-time in the Synchron runtime

foo : Int -> ()
foo val =
 let _ = syncT (sec 1) (usec 1) (send ledchan val) in
 foo (not val)

main = let _ = spawnExternal ledchan 1 in foo 1
```

In the above program, `foo` is the only software process, and there is one external hardware process for the LED driver that can be communicated with, using the `ledChan` channel. Line 11 is the critical part of the logic that sets the period of the program at 1 second, and the recursion at Line 12 keep the program alive forever. Appendix A shows the details of scheduling this program. We discuss a more involved example using `syncT` in Section 6.3.

4 Synchronisation Algorithms

The synchronous nature of message-passing is the foundation of the Synchron API. In this section, we describe the runtime algorithms, in an abstract form, that enable processes to synchronise. The Synchron runtime implements these algorithms, which drives the scheduling of the various software processes.

In Synchron, we synchronise on events. **Events**, in our API, fall into the categories of *base* events and *composite* events. The base events are `send` and `recv` and events created using `choose` are composite.

```
1 composite_event = choose (send c1 m1) (choose (send c2 m2) (send c3 m3))
```

From the API's point of view, composite events resemble a tree with base events in the leaves. However, for the algorithm descriptions here, we consider an event to be a *set* of base events. An implementation could impose an ordering on the base events that make up a composite event. Different orderings correspond to different event-prioritisation algorithms.

In the algorithm descriptions below, a **Channel** consists of two FIFO queues, one for `send` and one for `recv`. On these queues, process identities are stored. While blocked on a `recv` on a channel, that process' id is stored in the receive queue of that channel and likewise for `send` and the send-queue. Synchronous exchange of the message means that messages themselves do not need to be maintained on a queue.

Additionally, the algorithms below rely on there being a queue of processes that are ready to execute. This queue is called the `readyQ`. In the algorithm descriptions below, handling of `wrap` has been omitted. A function wrapped around an event specifies an operation that should be performed after synchronisation has been completed. Also, we abstract over the synchronisation of hardware events. As a convention, `self` used in the algorithms below refers to the process from which the `sync` operation is executed.

4.1 Synchronising events

The synchronisation algorithm, that performs the API operation `sync`, accepts a set of base events. The algorithm searches the set of events for a base event that has a sender or receiver blocked (ready to synchronize) and passes the message between sender and receiver. Algorithm 1 provides a high-level view of the synchronisation algorithm.

The first step in synchronisation is to see if there exists a synchronisable event in the set of base events. The *findSynchronisableEvent* algorithm is presented in Algorithm 2.

If the *findSynchronisableEvent* algorithm is unable to find an event that can be synchronised, the process initiating the synchronisation is blocked. The process identifier then gets added to all the channels involved in the base events of the set. This is shown in Algorithm 3.

After registering the process identifiers on the channels involved, the currently running process should yield its hold on the CPU, allowing another process to run. The next process to start running is found using the *dispatchNewProcess* algorithm in Algorithm 4.

When two processes are communicating, the first one to be scheduled will block as the other participant in the communication is not yet waiting on the channel. However, when

Algorithm 1 The synchronisation algorithm

```
Data: event : Set
ev ← findSynchronisableEvent(event);
if ev ≠ ∅ then
    syncNow(ev);
else
    block(event);
    dispatchNewThread();
end
```

Algorithm 2 The findSynchronisableEvent function

```
Data: event : Set
Result: A synchronisable event or ∅
foreach e ∈ event do
    if e.baseEventType == SEND then
        if ¬isEmpty(e.channelNo.recvq) then
            return e
        end
    else if e.baseEventType == RECV then
        if ¬isEmpty(e.channelNo.sendq) then
            return e
        end
    else return ∅;                                  /* Impossible case */
end
return ∅ ;                              /* No synchronisable event found */
```

Algorithm 3 The block function

```
Data: event : Set
foreach e ∈ event do
    if e.baseEventType == SEND then
        e.channelNo.sendq.enqueue(self);
    else if e.baseEventType == RECV then
        e.channelNo.recvq.enqueue(self);
    else Do nothing;                                /* Impossible case */
end
```

Algorithm 4 The dispatchNewProcess function

```
if readyQ ≠ ∅ then
    process ← dequeue(readyQ);
    currentProcess = process;
else
    relinquish control to the underlying OS
end
```

dispatchNewProcess dispatches the second process, the *findSynchronisableEvent* function will return a synchronisable event and the *syncNow* operation does the actual message passing. The algorithm of *syncNow* is given in Algorithm 5 below.

Algorithm 5 The syncNow function

```
Data: A base-event value - event
if event.baseEventType == SEND then
    receiver ← dequeue(event.channelNo.recvq);
    deliverMSG(self, receiver, msg) ;      /* pass msg from self to receiver */
    readyQ.enqueue(self);
else if event.baseEventType == RECV then
    sender ← dequeue(event.channelNo.sendq);
    deliverMSG(sender, self, msg) ;        /* pass msg from sender to self */
    readyQ.enqueue(sender);
else Do nothing;                                        /* Impossible case */
```

4.2 Timed synchronisation of events

Now, we look at the *timed* synchronisation algorithms. Timed synchronisation is handled by a two-part algorithm - the first part (Algorithm 6) runs when a process is executing the `syncT` API operation, and the second part (Algorithm 7) is executed later, after the baseline time specified in the timed synchronisation, `syncT`, call.

These algorithms rely on there being an alarm facility based on absolute wall-clock time, which invokes Algorithm 7 at a specific time. The alarm facility provides the operation `setAlarm` used in the algorithms below. The algorithms also require a queue, `waitQ`, to hold processes waiting for their baseline time-point.

The `handleAlarm` function in Algorithm 7 runs when an alarm goes off and, at that point, makes a process from the waitQ ready for execution. When the alarm goes off, there could be a process running already that should either be preempted by a timed process with a tight deadline or be allowed to run to completion in case its deadline is the tightest. The other alternative is that there is no process running and the process acquired from the waitQ can immediately be made active.

5 Implementation in SynchronVM

The algorithms of Section 4 are implemented within the Synchron runtime. The Synchron API and runtime are part of a larger programming platform that is the bytecode-interpreted virtual machine called SynchronVM, which builds on the work by Sarkar et al. [26].

The execution unit of SynchronVM is based on the Categorical Abstract Machine (CAM) [6]. CAM supports the cheap creation of closures, as a result of which SynchronVM can support a functional language quite naturally. CAM was chosen primarily for its simplicity and availability of pedagogical resources [14].

5.1 System Overview

Figure 4 shows the architecture of SynchronVM. The whole pipeline consists of three major components - (i) the frontend, (ii) the middleware and (iii) the backend.

Algorithm 6 The time function

Data: Relative Baseline = $baseline$, Relative Deadline = $deadline$
$T_{wakeup} = self.T_{local} + baseline$;
if $deadline == 0$ **then**
  $T_{finish} = Integer.MAX$; `/* deadline = 0 implies no deadline */`
else
  $T_{finish} = T_{wakeup} + deadline$;
end
$self.deadline = T_{finish}$;
$baseline_{absolute} = T_{absolute} + baseline$;
$deadline_{absolute} = T_{absolute} + baseline + deadline$;
$cond1 = T_{absolute} > deadline_{absolute}$;
$cond2 = (T_{absolute} \geq baseline_{absolute}) \&\& (T_{absolute} \leq deadline_{absolute})$;
$cond3 = baseline < SET\_ALARM\_AFTER$;
if $baseline == 0 \vee cond1 \vee cond2 \vee cond3$ **then**
  $readyQ.enqueue(currentThread)$;
  $dispatchNewProcess()$;
end
$setAlarm(T_{wakeup})$;
$waitQ.enqueue(self).orderBy(T_{wakeup})$;
$dispatchNewProcess()$;

Algorithm 7 The handleAlarm function

Data: Wakeup Interrupt
$timedProcess \leftarrow dequeue(waitQ)$;
$T_{now} = timedProcess.baseline$;
$timedProcess.T_{local} = T_{now}$;
if $waitQ \neq \varnothing$ **then**
  $timedProcess_2 \leftarrow peek(waitQ)$; `/* Does not dequeue */`
  $setAlarm(timedProcess_2.baseline)$;
end
if $currentProcess == \varnothing$; `/* No process currently running */`
then
  $currentProcess = timedProcess$;
else
  if $timedProcess.deadline < currentProcess.deadline$ **then**
    `/* Preempt currently running process */`
    $readyQ.enqueue(currentProcess)$;
    $currentProcess = timedProcess$;
  else
    `/* Schedule timed process to run after currentProcess */`
    $readyQ.enqueue(timedProcess)$;
    $currentProcess.T_{local} = T_{now}$; `/* Avoids too much time drift */`
  end
end

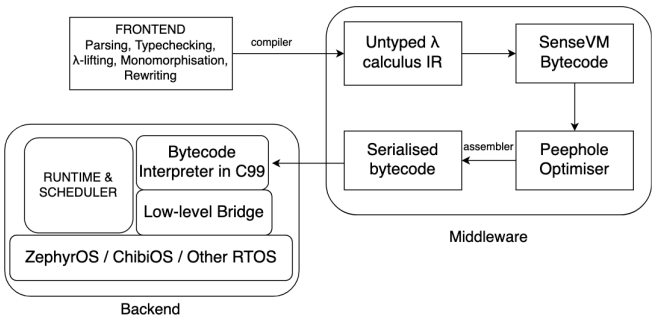


Figure 4 The compiler and runtime for SynchronVM

Frontend. We support a statically-typed, eagerly-evaluated, Caml-like functional language on the frontend. The language comes equipped with Hindley-Milner type inference. The polymorphic types are monomorphised at compile-time. The frontend module additionally runs a lambda-lifting pass to reduce the heap-allocation of closures.

Middleware. The frontend language gets compiled to an untyped lambda-calculus-based intermediate representation. This intermediate representation is then further compiled down to the actual bytecode that gets interpreted at run time. The generated bytecode emulates operations on a stack machine with a single environment register that holds the final value of a computation. This module generates specialised bytecodes that reduce the environment register-lookup using an operational notion called *r-free* variables described by Hinze [14]. On the generated bytecode, a peephole-optimisation pass applies further optimisations, such as β-reduction and *last-call optimisation* [14] (a generalisation of tail-call elimination).

Backend. The backend module is the principal component of the SynchronVM. It can be further classified into four components - (i) the bytecode interpreter, (ii) the runtime, (iii) a low-level *bridge* interface, and (iv) underlying OS/driver support.

Interpreter. The interpreter is written in C99 for portability. Currently, we use a total of 55 bytecodes operating on a stack machine.

Runtime. The runtime consists of a fixed-size stack with an environment register. A thread/process is emulated via the notion of a *context*, which holds a fixed-size stack, an environment register and a program counter to indicate the bytecode that is being interpreted. The runtime supports multiple but a *fixed* number of contexts, owing to memory constraints. A context holds two additional registers - one for the process-local clock (T_{local}) and the second to hold the deadline of that specific context (or thread).

The runtime has a garbage-collected heap to support closures and other composite values like tuples, algebraic data types, etc. The heap is structured as a list made of uniformly-sized tuples. For garbage collection, the runtime uses a standard *non-moving*, mark-and-sweep algorithm with the Hughes lazy-sweep optimisation [16].

Low-level Bridge. The runtime uses the low-level bridge interface functions to describe the various I/O-related interactions. It connects the runtime with the lowest level of the hardware. We discuss it in depth in Section 5.4.

Underlying OS/drivers. The lowest level of SynchronVM uses a real-time operating system that provides drivers for interacting with the various peripherals. The low-level is not restricted to use any particular OS, as shall be demonstrated in our examples using both the Zephyr-OS and ChibiOS.

5.1.1 Concurrency, I/O and Timing bytecode instructions

For accessing the operators of our programming interface as well as any general runtime-based operations, SynchronVM has a dedicated bytecode instruction - `CALLRTS n`, where `n` is a natural number to disambiguate between operations. Table 1 shows the bytecode operations corresponding to our programming interface.

spawn	CALLRTS 0
channel	CALLRTS 1
send	CALLRTS 2

recv	CALLRTS 3
sync	CALLRTS 4
choose	CALLRTS 5

spawnExternal	CALLRTS 6
wrap	CALLRTS 7
syncT	CALLRTS 8; CALLRTS 4

Table 1 Concurrency, I/O and Timing bytecodes

A notable aspect is the handling of the `syncT` operation, which gets compiled into a sequence of two instructions. The first instruction in the `syncT` sequence is `CALLRTS 8` which corresponds to Algorithm 6 in Section 4. When the process is woken up again as a result of Algorithm 7 it proceeds with the next instruction in the sequence which is `sync`, `CALLRTS 4`.

5.2 Message-passing with events

All forms of communication and I/O in SynchronVM operate via synchronous message-passing. However, a distinct aspect of SynchronVM's message-passing is the separation between the *intent* of communication and the actual communication. A value of type `Event` indicates the intent to communicate.

An event-value, like a closure, is a concrete runtime value allocated on the heap. The fundamental event-creation primitives are `send` and `recv`, which Reppy calls base-event constructors [25]. The event composition operators like `choose` and `wrap` operate on these base-event values to construct larger events. When a programmer attempts to send or receive a message, an event-value captures the channel number on which the communication was desired. When this event-value is synchronised (Section 4), we use the channel number as an identifier to match between prospective senders and receivers. Listing 6 shows the heap representation of an event-value as the type `event_t` and the information that the event-value captures on the SynchronVM.

Listing 6 Representing an `Event` in SynchronVM

```
typedef enum {
   SEND, RECV
} event_type_t;

typedef struct {
  event_type_t e_type; //  8 bits
  UUID channel_id;     //  8 bits
} base_evt_simple_t;

typedef struct {
  base_evt_simple_t  evt_details; // stored with 16 bits free
  cam_value_t wrap_func_ptr; // 32 bits
} base_event_t;


typedef struct {
  base_event_t bev; // 32 bits
  cam_value_t  msg; // 32 bits; NULL for recv
} cam_event_t;

typedef heap_index event_t;
```

An event is implemented as a linked list of base-events constructed by applications of the `choose` operation. Each element of the list captures (i) the message that is being sent or received, (ii) any function that is wrapped around the base-event using `wrap`, (iii) the channel being used for communication and (iv) an enum to distinguish whether the base-event is a `send` or `recv`. Fig 5 visualises an event upon allocation to the Synchron runtime's heap.

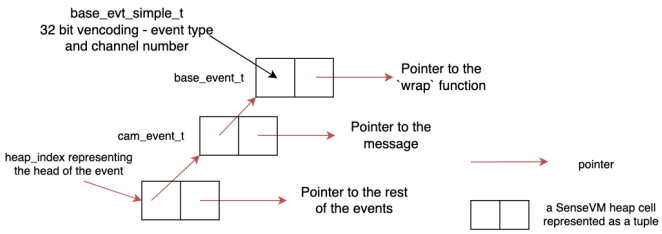


Figure 5 An event on the SynchronVM heap

The linked-list, as shown above, is the canonical representation of an `Event`-type. It can represent any complex composite event. For instance, if we take an example composite event that is created using the base-events, e_1, e_2, e_3 and a wrapping function wf_1, it can always be rewritten to its canonical form.

```
choose e1 (wrap (choose e2 e3) wf1)

-- Rewrite to canonical form --

choose e1 (choose (wrap e2 wf1) (wrap e3 wf1))
```

The `choose` operation can simply be represented as *cons*ing onto the event list.

5.3 The scheduler

SynchronVM's scheduler is a hyrbid of cooperative and preemptive scheduling. For applications that do not use `syncT`, the scheduler is cooperative in nature. Initially the threads are scheduled in the order that the main method calls them. For example,

```
main =
 let _ = spawn thread1 in
 let _ = spawn thread2 in
 let _ = spawn thread3 in ...
```

The scheduler orders the above in the order of `thread1` first, `thread2` next and `thread3` last. As the program proceeds, the scheduler relies on the threads to yield their control according to the algorithms of Section 4. When the scheduler is unable to find a matching thread for the currently running thread that is ready to synchronise the communication, it blocks the current thread and calls the *dispatchNewThread*() function to run other threads (see Algorithm 1). On the other hand, when synchronisation succeeds, the scheduler puts the message-sending thread in the `readyQ` and the message-receiving thread starts running.

The preemptive behaviour of the scheduler occurs when using `syncT`. For instance, when a particular *untimed* thread is running and the baseline time of a timed thread has arrived, the scheduler then preempts the execution of the *untimed* thread and starts running the timed thread. A similar policy is also observed when the executing thread's deadline is later than a more urgent thread; the thread with the earliest deadline is chosen to be run at that instance. Algorithm 7 shows the preemptive components of the scheduler.

The SynchronVM scheduler also handles hardware driver interactions via message-passing. The structure that is used for messaging is shown below:

Listing 7 A SynchronVM hardware message

```
typedef struct {
  uint32_t sender_id;
  uint32_t msg_type;
  uint32_t data;
  Time timestamp;
} svm_msg_t;
```

The `svm_msg_t` type contains a unique sender id for each driver that is the same as the number used in `spawnExternal` to identify that driver. The 32 bit `msg_type` field can be used to specify different meanings for the next field, the `data`. The `data` is a 32 bit word. The `timestamp` field of a message struct is a 64 bit entity, explained in detail in Section 5.5.

When the SynchronVM scheduler has all threads blocked, it uses a function pointer called `blockMsg`, which is passed to it by the OS that starts the scheduler, to wait for any interrupts from the underlying OS (more details in Section 5.4). Upon receiving an interrupt, the scheduler uses the SynchronVM runtime's `handleMsg` function to handle the corresponding message. The function internally takes the message and unblocks the thread for which the message was sent. The general structure of SynchronVM's scheduler is shown in Algorithm 8.

Algorithm 8 The SynchronVM scheduler

Data: $blockMsg$ function pointer
$\forall threads\ set\ T_{local} = T_{absolute}$;
$svm\_msg\_t\ msg$;
while *True* **do**
  if *all threads blocked* **then**
    $blockMsg(\&msg)$; /* Relinquish control to OS */
    $handleMsg(msg)$;
  else
    $interpret(currentThread.PC)$;
  end
end

The T_{local} clock is initialised for each thread when starting up the scheduler. Also notable is the `blockMsg` function that relinquishes control to the underlying OS, allowing it to save power. When the interrupt arrives, the `handleMsg` function unblocks certain thread(s) so that when the $if..then$ clause ends, in the following iteration the *else* clause is executed and bytecode interpretation continues. We next discuss the low-level bridge connecting Synchron runtime to the underlying OS.

5.4 The Low-Level Bridge

The low-level bridge specifies two interfaces that should be implemented when writing peripheral drivers to use with SynchronVM. The first contains functions for reading and writing data synchronously to and from a driver. The second is geared towards interrupt-based drivers that asynchronously produce data.

The C-`struct` below contains the interface functions for reading and writing data to a driver as well as functions for checking the availability of data.

```
typedef struct ll_driver_s{
  void *driver_info;
  bool is_synchronous;
  uint32_t (*ll_read_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_write_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_data_readable_fun)(struct ll_driver_s* this);
  uint32_t (*ll_data_writeable_fun)(struct ll_driver_s* this);

  UUID channel_id;
} ll_driver_t;
```

The `driver_info` field in the `ll_driver_t` struct can be used by a driver that implements the interface to keep a pointer to lower-level driver specific data. For interrupt-based drivers, this data will contain, among other things, an *OS interoperation* struct. These OS interoperation structs are explained further below. A boolean indicates whether the driver is synchronous or not. Next the struct contains function pointers to the low-level driver's implementation of the interface. Lastly, a `channel_id` identifies the channel along which the driver is allowed to communicate with processes running on top of SynchronVM.

The `ll_driver_t` struct contains all the data associated with a driver's configuration in one place and defines a set of platform and driver independent functions for use in the runtime system, shown below:

```
uint32_t ll_read(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_write(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_data_readable(ll_driver_t *drv);
uint32_t ll_data_writeable(ll_driver_t *drv);
```

The OS interoperation structs, mentioned above, are essential for drivers that asynchronously produce data. We show their Zephyr and the ChibiOS versions below:

```
typedef struct zephyr_interop_s {
  struct k_msgq *msgq;
  int (*send_message)(struct zephyr_interop_s* this, svm_msg_t msg);
} zephyr_interop_t;
```

```
typedef struct chibios_interop_s {
  memory_pool_t *msg_pool;
  mailbox_t *mb;
  int (*send_message)(struct chibios_interop_s* this, svm_msg_t msg);
} chibios_interop_t;
```

In both cases, the struct contains the data that functions need to set up low-level message-passing between the driver and the OS thread running the SynchronVM runtime. Zephyr provides a message-queue abstraction that can take fixed-size messages, while ChibiOS supports a mailbox abstraction that receives messages that are the size of a pointer. Since ChibiOS mailboxes cannot receive data that is larger than a 32-bit word, a memory pool of messages is employed in that case. The structure used to send messages from the drivers is the already-introduced `svm_msg_t` struct, given in Listing 7.

The scheduler also needs to handle alarm interrupts from the wall-clock time subsystem, arising from the `syncT` operation. The next section discusses that component of SynchronVM.

5.5 The wall-clock time subsystem

Programs running on SynchronVM that make use of the timed operations rely on there being a monotonically increasing timer. The wall-clock time subsystem emulates this by implementing a 64bit timer that would take almost 7000 years to overflow at 84MHz frequency or about 36000 years at 16MHz. The timer frequency of 16MHz is used on the NRF52 board, while the timer runs at 84MHz on the STM32.

SynchronVM requires the implementation of the following functions for each of the platforms (such as ChibiOS and Zephyr) that it runs on :

```
bool      sys_time_init(void *os_interop);
Time      sys_time_get_current_ticks(void);
uint32_t  sys_time_get_alarm_channels(void);
uint32_t  sys_time_get_clock_freq(void);
bool      sys_time_set_wake_up(Time absolute);
Time      sys_time_get_wake_up_time(void);
bool      sys_time_is_alarm_set(void);
```

The timing subsystem uses the same OS interoperation structs as drivers do and thus has access to a communication channel to the SynchronVM scheduler. The interoperation is provided to the subsystem at initialisation using `sys_time_init`.

The key functionality implemented by the timing subsystem is the ability to set an alarm at an absolute 64-bit point in time. Setting an alarm is done using `sys_time_set_wake_up`. The runtime system can also query the timing subsystem to check if an alarm is set and at what specific time.

The low-level implementation of the timing subsystem is highly platform dependent at present. But on both Zephyr and ChibiOS, the implementation is currently based on a single 32-bit counter configured to issue interrupts at overflow, where an additional 32-bit value is incremented. Alarms can only be set on the lower 32-bit counter at absolute 32-bit values. Additional logic is needed to translate between the 64-bit alarms set by SynchronVM and the 32-bit timers of the target platforms. Each time the overflow interrupt happens, the interrupt service routine checks if there is an alarm in the next 32-bit window of time and in that case, enables a compare interrupt to handle that alarm. When the alarm interrupt happens, a message is sent to the SynchronVM scheduler in the same way as for interrupt based drivers, using the message queue or mailbox from the OS interoperation structure.

Revisiting the requirements for implementing `syncT`, we find that our scheduler (1) provides a preemptive mechanism to override the fair scheduling, (2) has access to a wall-clock time source, and (3) implements an earliest-deadline-first scheduling policy that attempts to match the local time and the absolute time.

5.6 Porting SynchronVM to another RTOS

For porting SynchronVM to a new RTOS, one needs to implement - (1) the wall-clock time subsystem interface from Section 5.5, (2) the low-level bridge interface (Section 5.4) for each peripheral, and (3) a mailbox or message queue for communication between asynchronous drivers and the runtime system, required by the time subsystem.

Our initial platform of choice was ZephyrOS for its platform-independent abstractions. The first port of SynchronVM was on ChibiOS, where the wall-clock time subsystem was 254 lines of C-code. The drivers for LED, PWM, and DAC were about 100 lines of C-code each.

6 Case Studies

Finite-State Machines with Synchron

We will begin with two examples of expressing state machines (involving callbacks) in the Synchron API. Our examples are run on the `NRF52840DK` microcontroller board containing four buttons and four LEDs. We particularly choose the button peripheral because its drivers have a callback-based API that typically leads to non-linear control-flows in programs.

6.1 Four-Button-Blinky

We build on the *button-blinky* program from Listing 4 presented in Section 3.2. The original program, upon a single button-press, would light up an LED corresponding to that press and switch off upon the button release. We now extend that program to produce a one-to-one mapping between four LEDs and four buttons such that button1 press lights up LED1, button2 lights up LED2, button3 lights up LED3 and button4 lights up LED4 (while the button releases switch off the corresponding LEDs).

The state machine of button-blinky is a standard two-state automaton that moves from the ON-state to OFF on button-press and vice versa. Now, for the four button-LED combinations, we have four state machines. We can combine them using the `choose` operator.

Listing 8 shows the important parts of the logic. The four state machines are declared in Lines 1 to 4, and their composition happens in Line 6 using the `choose` operator. See Appendix B.1 for the full program.

Listing 8 The Four-Button-Blinky program expressed in the Synchron API

```
1 press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
2 press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
3 press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
4 press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))
5
6 anybutton = choose press1 (choose press2 (choose press3 press4))
7
8 program : ()
9 program = let _ = sync anybutton in program
```

6.2 A more intricate FSM

We now construct a more intricate finite-state machine involving intermediate states that can move to an error state if the desired state-transition buttons are not pressed. For this example a button driver needs to be configured to send only one message per button press-and-release. So there is no separate button-on and button-off signal but one signal per button.

In this FSM, we glow the LED1 upon consecutive presses of button1 and button2. We use the same path to turn LED1 off. However, if a press on button1 is followed by a press of button 1 or 3 or 4, then we move to an error state indicated by LED3. We use the same path to switch off LED3. In a similar vein, consecutive presses of button3 and button4 turns on LED2 and button3 followed by button 1 or 2 or 3 turns on the error LED - LED3. Fig. 6 shows the FSM diagram of this application, omitting self-loops in the OFF state.

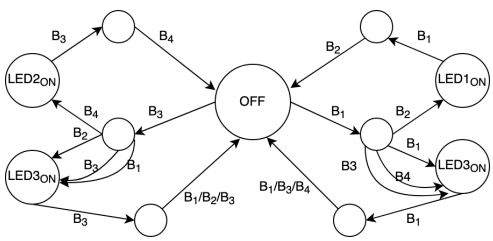


Figure 6 A complex state machine

Listing 9 shows the central logic expressing the FSM of Fig 6 in the Synchron API (See Appendix B.2 for the full program). This FSM can be viewed as a composition of two

separate finite state machines, one on the left side of the OFF state involving LED2 and LED3 and one on the right side involving LED1 and LED3. Once again, we use the `choose` operator to compose these two state machines.

Listing 9 The complex state machine running on the SynchronVM

```
errorLed x = ledchan3

fail1ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan3) errorLed)
                 (wrap (recv butchan4) errorLed))

fail2ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan2) errorLed)
                 (wrap (recv butchan3) errorLed))

led1Handler x =
   sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)

led2Handler x =
   sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)

led : Int -> ()
led state =
  let fsm1 = wrap (recv butchan1) led1Handler in
  let fsm2 = wrap (recv butchan3) led2Handler in
  let ch = sync (choose fsm1 fsm2) in
  let _ = sync (send ch (not state)) in
  led (not state)
```

In Listing 9, the `led1Handler1` and `ledHandler2` functions capture the intermediate states after one button press, when the program awaits the next button press. The error states are composed using the `choose` operator in the functions `fail1ev` and `fail2ev`.

The compositional nature of our framework is visible in line no. 21 where we compose the two state machines, `fsm1` and `fsm2`, using the `choose` operator. Synchronising on this composite event returns the LED channel (demonstrating a higher-order approach) on which the process should attempt to write. This program is notably a highly callback-based, reactive program that we have managed to represent in an entirely synchronous framework.

6.3 A soft-realtime music playing example

We present a soft-realtime music playing exercise from a Real-Time Systems course, expressed using the Synchron API. We choose the popular nursery rhyme - "Twinkle, Twinkle, Little Star". The program plays the tune repeatedly until it is stopped.

The core logic of the program involves periodically writing a sequence of 1's and 0's to a DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Our sound is generated at the 196Hz G3 music key.

Listing 10 shows the principal logic of the program expressed using the Synchron API. Note that we use `syncT` to describe a new temporal combinator `after` that determines the periodicity of this program. The list `twinkle` (line 17) holds the 28 notes in the twinkle song and the list `durations` (line 18) provides the length of each note. Appendix B provides full details of the program.

Listing 10 The *Twinkle, Twinkle* tune expressed using the Synchron API

```
msec t = t * 1000
usec t = t
```

```
after t ev = syncT t 0 ev

-- note frequencies
g = usec 2551
a = usec 2273
b = usec 2025
c = usec 1911
d = usec 1703
e = usec 1517

hn = msec 1000 -- half note
qn = msec  500 -- quarter note

twinkle, durations : List Int
twinkle   = [ g,  g,  d,  d,  e,  e,  d.... ] -- 28 notes
durations = [qn, qn, qn, qn, qn, qn, hn.... ]

dacC  = channel ()
noteC = channel ()

playerP : List Int -> List Int -> Int -> () -> ()
playerP melody nt n void =
  if (n == 29)
  then let _ = after (head nt) (send noteC (head twinkle)) in
       playerP (tail twinkle) durations 2 void
  else let _ = after (head nt) (send noteC (head melody)) in
       playerP (tail melody) (tail nt) (n + 1) void

tuneP : Int -> Int -> () -> ()
tuneP timePeriod vol void =
  let newtp =
      after timePeriod (choose (recv noteC)
                               (wrap (send dacC (vol * 4095))
                                     (λ _ -> timePeriod))) in
  tuneP newtp (not vol) void

main =
  let _ = spawnExternal dacC 0 in
  let _ = spawn (tuneP (head twinkle) 1) in
  let _ = spawn (playerP (tail twinkle) durations 2) in()
```

The application consists of two software processes and one external hardware process. We use two channels - `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes. Looking at what each software process is doing -

playerP. This process runs at the rate of a note's length. For a quarter note it wakes up after 500 milliseconds (1000 msecs for a half note), traverses the next element of the `twinkle` list and sends it along the `noteC` channel. It circles back after completing all 28 notes.

tuneP. This process is responsible for actually creating the sound. Its running rate varies depending on the note that is being played. For instance, when playing note C, it will write to the DAC at at a rate of 1911 microseconds-per-write. However, upon receiving a new value along `noteC`, it changes its write frequency to the new value resulting in changing the note of the song.

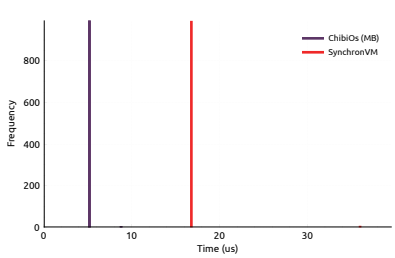
7 Benchmarks

7.1 Interpretive overhead measurements

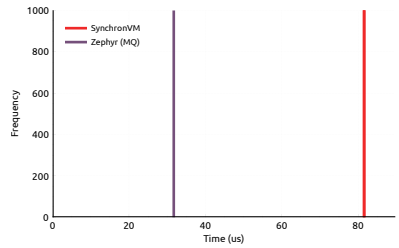
We characterise the overhead of executing programs on top of SynchronVM, compared to running them directly on either Zephyr or ChibiOS, by implementing *button-blinky* directly on top of these operating systems and measuring the response-time differences.

The button-blinky program copies the state of a button onto an LED, something that could be done very rapidly at a large CPU utilization cost by continuously polling the button state and writing it to the LED. Instead, the Zephyr and ChibiOS implementations are interrupt-based and upon receiving a button interrupt (for either button press or release), send a message to a thread that is kept blocking until such messages arrive. When the thread receives a message indicating a button down or up, it sets the LED to on or off. This approach keeps the low-level implementation in Zephyr and ChibiOS similar to SynchronVM and indicates the interpretive and other overheads in SynchronVM.

The data in charts presented here is collected using an STM32F4 microcontroller based testing system connected to either the NRF52 or the STM32F4 system under test (SUT). The testing system provides the stimuli, setting the GPIO (button) to either active or inactive and measures the time it takes for the SUT to respond on another GPIO pin (symbolising the LED). The testing system connects to a computer displaying a GUI and generates the plots used in this paper. Each plot places measured response times into buckets of similar time, and shows the number of samples falling in each bucket as a vertical bar. Each bucket is labelled with the average time of the samples it contains.



(a) Response time comparison between a C-code implementation using ChibiOS against the same program on SynchronVM (running on ChibiOS). Data obtained on the STM32F4 microcontroller.



(b) Response time comparison between a C-code implementation using the Zephyr OS against the same program on SynchronVM (running on Zephyr). Data obtained on the NRF52 microcontroller.

Figure 7 Button-blinky response times comparison between C and SynchronVM

Fig. 7a shows the SynchronVM response time in comparison to the implementation of the program running on ChibiOS using its mailbox abstraction (MB). There the overhead is about 3x. Fig. 7b compares response times for SynchronVM and the Zephyr message queue based implementation (MQ), and shows an overhead of 2.6x.

In the measurements relative to ChibiOS (Figure 7a), there are outliers both when running on ChibiOS directly and when running SynchronVM on top of ChibiOS. In the ChibiOS (MB) data there are 6 outliers where a response takes 1.6 times longer than an average non-outlier response. For SynchronVM, 8 outliers take 2.1 times longer than an average non-outlier SynchronVM response.

7.2 Effects of Garbage Collection

This experiment measures the effects of garbage collection on response time by repeatedly running the 10000 samples test for different heap-size configurations of SynchronVM. A smaller heap should lead to more frequent interactions with the garbage collector, and the effects of the garbage collector on the response time should magnify.

As a smaller heap is used, the number of outliers should increase if the outliers are due to garbage collection. The following table shows the number of outliers at each size configuration for the heap used, and there is an indication that GC is the cause of outliers.

Heap size (bytes)	256	512	1024	2048	4096	8192
Outliers NRF52 on Zephyr	3334	1429	811	491	0	81
Outliers STM32 on ChibiOs	3339	1430	810	491	0	80

Figures 8 and 9 show the response-time numbers across the heap sizes of 8192, 4096, 2048, 1024, 512 and 256 Bytes. A general observable trend is that as the heap size decreases and GC increases, the response time numbers hover towards the farther end of the X-axis. This trend is most visible for the heap size of 256 bytes, which is our smallest heap size. Note that we cannot collect enough sample data for response-time if we switch off the garbage collector (as a reference value), as the program would very quickly run out of memory and terminate.

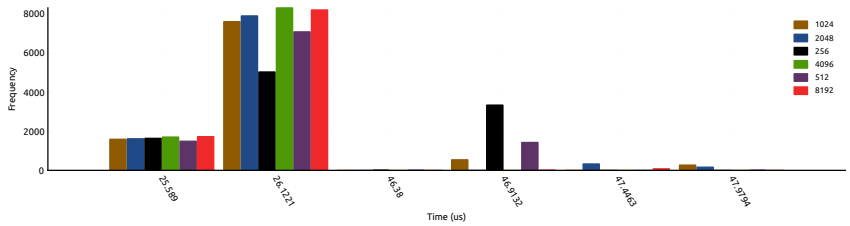


Figure 8 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the STM32F4 microcontroller running SynchronVM on top of ChibiOS. Each bucket size is approx 0.533us. Uses 10000 samples.

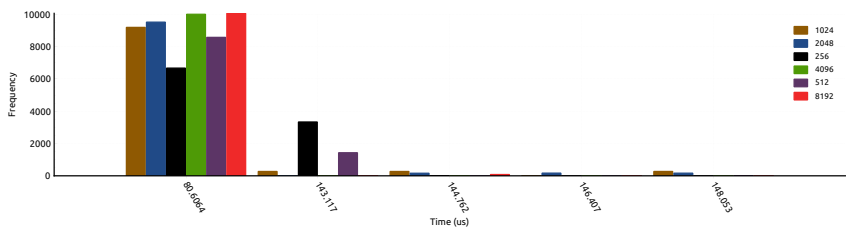


Figure 9 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the NRF52 microcontroller running SynchronVM on top of the Zephyr OS. Each bucket size is approx 1.65us. Uses 10000 samples.

7.3 Power Usage

Fig. 10 shows the power usage of the NRF52 microcontroller running the button-blinky program for three implementations. The first implementation is a polling version of the program in C. The second program uses a callback-based version of button-blinky [10].

The last program is Listing 4 running on SynchronVM. The measurements are made using the Ruideng UM25C ammeter. We collect momentary readings from the ammeter after the value has stabilised.

Notable in Fig. 10 is the polling-based C implementation's use of 0.0175 Watts of power in a button-off state, whereas SynchronVM consumes five times less power (0.0035 Watts). This is comparable to the callback-based C implementation's use of 0.003 Watts. Integrating the power usage over time will likely make the difference between SynchronVM and the callback-based C version more noticeable. However, the simplicity and declarative nature of the Synchron-based code is a good tradeoff.

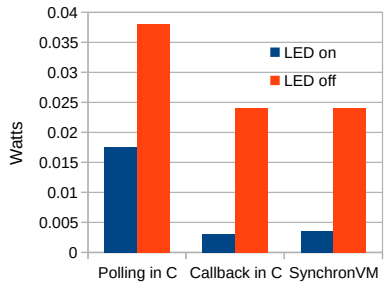


Figure 10 Power usage measured on the NRF52 microcontroller

7.4 Memory Footprint

SynchronVM resides on the Flash memory of a microcontroller. On Zephyr, a tiny C application occupies 17100 bytes, whereas the same SynchronVM application occupies 49356 bytes, which gives the VM's footprint as 32256 bytes. For ChibiOS, the C application takes 18548 bytes, while the SynchronVM application takes 53868 bytes. Thus, SynchronVM takes 35320 bytes in this case. Hence, we can estimate SynchronVM's rough memory footprint at 32 KB, which will grow with more drivers.

7.5 Precision and Jitter

Jitter can be defined as the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal. We want to evaluate how our claims of `syncT` reducing jitter pans out in practice.

Listing 11 below is written in a naive way to illustrate how jitter manifests in programs. Figure 11a shows what the oscilloscope draws, set to persistent mode drawing while sampling the signal from the Raspberry Pi outputs.

The Raspberry Pi program reads the status of a GPIO pin and then inverts its state back to that same pin. The program then goes to sleep using `usleep` for 400us. The goal frequency was 1kHz and sleeping for 400us here gave a roughly 1.05kHz signal. The more expected sleep time of 500us to generate a 1kHz signal led, instead, to a much lower frequency. So, the 400us value was found experimentally.

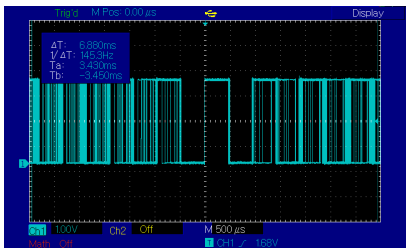
```
while (1) {
 uint32_t state = GPIO_READ(23);
 if (state) {
   GPIO_CLR(23);
 } else {
   GPIO_SET(23);
 }
 usleep(400);
 }
// main method and other setup
    elided
```

Listing 11 Raspberry Pi C code

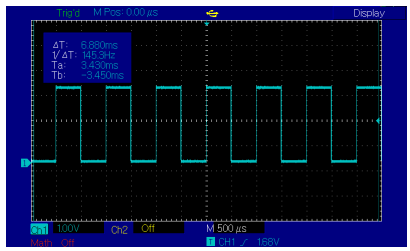
```
ledchan = channel ()

foo : Int -> ()
foo val =
 let _ = syncT 500 0 (send
    ledchan val)
  in foo (not val)

main =
 let _ = spawnExternal ledchan 1
  in foo 1
```

Listing 12 SynchronVM 1KHz wave code

Listing 12 shows the same 1kHz frequency generator for SynchronVM. Note that, in this case, specifying a baseline of 500us leads to a 1kHz wave (compared to the 400us used above that together with additional delays of the system gave a roughly 1kHz wave).



(a) Illustrating the amount of jitter on the square wave generated from the Raspberry Pi by setting the oscilloscope display in persistent mode.



(b) A 1kHz square wave generated using SynchronVM running on the STM32F4 with no jitter

Figure 11 A 1 kHz frequency generator on the Raspberry PI (in C) and STM32 (Synchron)

7.6 Load Test

The SynchronVM program in the previous section could produce a 1kHz-wave with no jitter. However, the only operation that the program did was produce the square wave. In this section, we want to test how much computational load can be performed by Synchron while producing the square wave. We emulate the workload using the following program.

```
21 load i n =
22  let _ = fib_tailrec n in
23  let _ = syncT 8000 0 (send
      ledchan i)
24   in load (not i)
```

```
25 loop i a b n =
26  if i == n then a
27  else loop (i+1) (b) (a+b) n
28
29 fib_tailrec n = loop 0 0 1 n
```

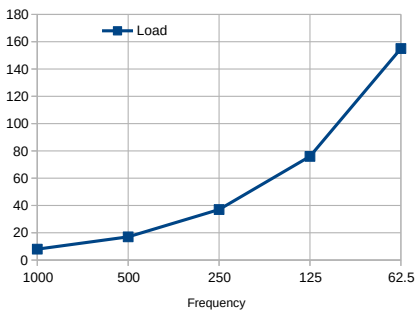


Figure 12 Load testing SynchronVM with the nth fibonacci number function

At a given frequency, it is possible to calculate only up to a certain Fibonacci number while generating the square wave at the desired frequency. For example, when generating a 62.5 Hz wave, it is only possible to calculate up to the 155th Fibonacci number. If the 156th number is calculated, the wave frequency drops below 62.5 Hz.

Fig. 12 plots the nth Fibonacci numbers that can be calculated against the square wave frequencies that get generated without jitters. Our implementation of `fib_tailrec` involves 2 addition operations, 1 equality comparison and 1 recursive call. So, calculating the 155th Fibonacci number involves $155 * 4 = 620$ major operations. The trend shows that the load capacity of SynchronVM grows linearly as the desired frequency of the square wave is halved.

7.7 Music Program Benchmarks

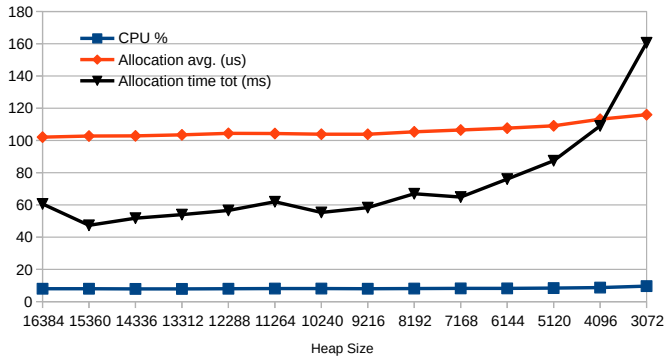


Figure 13 CPU usage and allocation trends over a 1 minute window for Listing 10

We now provide some benchmarks on the music program from Section 6.3. Figure 13 shows CPU usage, average time it takes to allocate data and total time spent doing allocations in a 1 minute window. The values used in the chart come from the second minute of running the music application. The values from the first minute of execution are discarded as those would include the startup phase of the system. The amount of heap made available to the runtime system is varied from a roomy 16384 bytes down to 3072 bytes.

The sweep phase of our garbage collector is intertwined with the allocations phase. Hence, instead of showing the GC time, the chart shows statistics related to all allocations that take a measurable amount of time using the ChibiOS 10KHz system timer. All allocations taking less than 100us are left of out of the statistics (and not counted towards averaging).

The data in Fig. 13 shows that CPU usage of the music application is pretty stable at around 8 percent over the one minute window. It increases slightly for the very small heap sizes and ends up at nearly 10 percent at the smallest heap size that can house the program.

In terms of allocation, the average time of an allocation (in usecs) increases when the probability of a more expensive allocation increases, which in turn increases with small heap sizes. In the last data series, the total amount of time spent in allocations (in msecs) grows considerably as the heap size drops below 7168 bytes - an indicator of increased GC activity.

7.8 Discussion

Our benchmarks, so far, show promising results for power, memory, and CPU usage. However, SynchronVM's response time is 2-3x times slower than native C code, which needs improvement. We attribute the slowness to the CAM-based execution engine, which we hope to mitigate by moving to a ZAM-based machine [18].

Our synthetic load test (Fig. 12) indicates that the VM can support around 150 operations for applications that operate around 250Hz (such as balance bots). Our music program falls in the range of 200-500 Hz, and SynchronVM could sustain that frequency without introducing any jitter. There exist other non-periodic applications with much lesser frequencies, such as sensor networks, where SynchronVM could be applicable.

Comparison with Asynchronous Message-Passing

The Synchron API chooses a synchronous message-passing model, in contrast, with actor-based systems like Erlang that support an asynchronous message-passing model with each process containing a mailbox. We believe that a synchronous message-passing policy is better suited for embedded systems for the following reasons:

1. Embedded systems are highly memory-constrained, and asynchronous send semantics assume the *unboundedness* of an actor's mailbox, which is a poor assumption in the presence of memory constraints. Once the mailbox becomes full, message-sending becomes blocking, which is already the default semantics of synchronous message-passing.
2. Acknowledgement is implicit in synchronous message-passing systems, in contrast to explicit message acknowledgement in asynchronous systems that leads to code bloat. Additionally, if a programmer forgets to remove acknowledgement messages from an actor's mailbox, it leads to memory leaks.

8 Limitations and Future Work

In this section, we propose future work to improve the Synchron API and runtime.

8.1 Synchron API limitation

Deadline miss API. Currently, the Synchron API cannot represent actions that should happen if a task were to miss its deadline. We envision adapting the negative acknowledgement API of CML to represent missed-deadline handlers for Synchron.

8.2 SynchronVM limitations

Memory management. A primary area of improvement is upgrading our stop-the-world mark and sweep garbage collector and investigating real-time garbage collectors like Schism [23]. Another relevant future work would be investigating static memory-management schemes like regions [30] and techniques combining regions with GC [13].

Interpretation overhead. A possible approach to reducing our interpretation overhead could be pre-compiling our bytecode to machine code (AOT compilation). Similarly, dynamic optimization approaches like JITing could be an area of investigation.

Priority inversions. Although TinyTimber-style dynamic priorities might reduce priority inversion occurrences, they can still occur on the SynchronVM. Advanced approaches like priority inheritance protocols [27] need to be experimented with on our scheduler.

8.3 General runtime limitations

- Power efficiency and lifetime while operating from a small battery is challenging for a byte-code interpreting virtual machine.
- Safety-critical, hard real-time systems remain out of reach with a bytecode-interpreted and garbage collected virtual machine.

9 Related Work

Among functional languages running on microcontrollers, there exists OCaml running on OMicroB [32], Scheme running on Picobit [29] and Erlang running on AtomVM [4]. Synchron

differs from these projects in the aspect that we identify certain fundamental characteristics of embedded systems and accordingly design an API and runtime to address those demands. As a result, our programming interface aligns more naturally to the requirements of an embedded systems application, in contrast with general-purpose languages like Scheme.

The Medusa [2] language and runtime is the inspiration behind our uniform framework of concurrency and I/O. Medusa, however, does not provide any timing based APIs, and their message-passing framework is based on the actor model (See Section 7.8).

In the real-time space, a safety-critical VM that can provide hard real-time guarantees on Real-Time Java programs is the FijiVM [24] implementation. A critical innovation of the project was the Schism real-time garbage collector [23], from which we hope to draw inspiration for future work on memory management.

RTMLton [28] is another example of a real-time project supporting a general-purpose language like SML. RTMLton adapts the MLton runtime [34] with ideas from FijiVM to enable handling real-time constraints in SML. CML is available as an SML library, so RTMLton provides access to the event framework of CML but lacks the uniform concurrency-I/O model and the `syncT` operator of Synchron.

The Timber language [5] is an object-oriented language that inspired the `syncT` API of Synchron. Timber was designed for hard real-time scenarios; related work on estimating heap space bounds [17] could perhaps benefit our future research.

The WebAssembly project (WASM) has spawned sub-projects like WebAssembly Micro Runtime (WAMR) [1] that allows running languages that compile to WASM on microcontrollers. Notable here is that while several general-purpose languages like JavaScript can execute on ARM architectures by compiling to WebAssembly, they lack the native support for the concurrent, I/O-bound, and timing-aware programs that is naturally provided by our API and its implementation. Reactive extensions of Javascript, like HipHop.js [3], are being envisioned to be used for embedded systems.

Another related line of work is embedding domain-specific languages like Ivory [9] and Copilot [22] in Haskell to generate C programs that can run on embedded devices. This approach differs from ours in the aspect that two separate languages dictate the programming model of an EDSL - the first being the DSL itself and the second being the host language (Haskell). We assess that having a single language (like in Synchron) provides a more uniform programming model to the programmer. However, code-generating EDSLs have very little runtime overheads and, when fully optimised, can produce high performance C.

10 Conclusion

In this paper, we have presented Synchron - an API and runtime for embedded systems, which we implement within the larger SynchronVM. We identified three essential characteristics of embedded applications, namely being concurrent, I/O–bound, and timing-aware, and correspondingly designed our API to address all three concerns. Our evaluations, conducted on the STM32 and NRF52 microcontrollers, show encouraging results for power, memory and CPU usage of the SynchronVM. Our response time numbers are within the range of 2-3x times that of native C programs, which we envision being improved by moving to a register-based execution engine and by using smarter memory-management strategies. We have additionally demonstrated the expressivity of our API through state machine-based examples, commonly found in embedded systems. Finally, we illustrated our timing API by expressing a soft real-time application, and we expect further theoretical investigations on the worst-case execution time and schedulability analysis on SynchronVM.

References

1 WAMR - WebAssembly Micro Runtime, 2019. URL: https://github.com/bytecodealliance/wasm-micro-runtime.

2 Thomas W. Barr and Scott Rixner. Medusa: Managing Concurrency and Communication in Embedded Systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 439–450. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/barr.

3 Gérard Berry and Manuel Serrano. Hiphop.js: (A)Synchronous reactive web programming. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 533–545. ACM, 2020. doi:10.1145/3385412.3385984.

4 Davide Bettio. AtomVM, 2017. URL: https://github.com/bettio/AtomVM.

5 Andrew P Black, Magnus Carlsson, Mark P Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, OGI School of Science and Engineering, Oregon Health and Sciences University, Technical Report CSE 02-002. April 2002, 2002.

6 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 1985. doi:10.1007/3-540-15975-4_29.

7 Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The Synchronous Hypothesis and Synchronous Languages. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. doi:10.1201/9781420038163.ch8.

8 Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Andrew T. Campbell, Philippe Bonnet, and John S. Heidemann, editors, *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, pages 29–42. ACM, 2006. doi:10.1145/1182807.1182811.

9 Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. Guilt free ivory. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 189–200. ACM, 2015. doi:10.1145/2804302.2804318.

10 Zephyr examples. Zephyr button blinky, 2021. URL: https://pastecode.io/s/szpf673u.

11 The Linux Foundation. Zephyr RTOS. https://www.zephyrproject.org/. Accessed 2021-11-28.

12 Damien George. Micropython, 2014. URL: https://micropython.org/.

13 Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining Region Inference and Garbage Collection. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 141–152. ACM, 2002. doi:10.1145/512529.512547.

14 Ralf Hinze. The Categorical Abstract Machine: Basics and Enhancements. Technical report, University of Bonn, 1993.

15 C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.

16 R John M Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.

17 Martin Kero, Pawel Pietrzak, and Johan Nordlander. Live Heap Space Bounds for Real-Time Systems. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010. doi:10.1007/978-3-642-17164-2_20.

18 Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA, 1990.

19 Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. TinyTimber, Reactive Objects in C for Real-Time Embedded Systems. In *2008 Design, Automation and Test in Europe*, pages 1382–1385, 2008. doi:10.1109/DATE.2008.4484933.

20 Tommi Mikkonen and Antero Taivalsaari. Web Applications - Spaghetti Code for the 21st Century. In Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye, editors, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, pages 319–328. IEEE Computer Society, 2008. doi:10.1109/SERA.2008.16.

21 Johan Nordlander. *Programming with the TinyTimber kernel*. Luleå tekniska universitet, 2007.

22 Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A Hard Real-Time Runtime Monitor. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010. doi:10.1007/978-3-642-16612-9_26.

23 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 146–159. ACM, 2010. doi:10.1145/1806596.1806615.

24 Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In M. Teresa Higuera-Toledano and Martin Schoeberl, editors, *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ACM International Conference Proceeding Series, pages 110–119. ACM, 2009. doi:10.1145/1620405.1620421.

25 John H. Reppy. Concurrent ML: Design, Application and Semantics. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993. doi:10.1007/3-540-56883-2_10.

26 Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, and Mary Sheeran. Higher-Order Concurrency for Microcontrollers. In Herbert Kuchen and Jeremy Singer, editors, *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, pages 26–35. ACM, 2021. doi:10.1145/3475738.3480716.

27 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

28 Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. RTMLton: An SML Runtime for Real-Time Systems. In Ekaterina Komendantskaya and Yanhong Annie Liu, editors, *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, volume 12007 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2020. doi:10.1007/978-3-030-39197-3_8.

29 Vincent St-Amour and Marc Feeley. PICOBIT: A Compact Scheme System for Microcontrollers. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

30 Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

31 Hideyuki Tokuda, Clifford W. Mercer, Yutaka Ishikawa, and Thomas E. Marchok. Priority Inversions in Real-Time Communication. In *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*, pages 348–359. IEEE Computer Society, 1989. doi:10.1109/REAL.1989.63587.

32 Steven Varoumas, Benoît Vaugon, and Emmanuel 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.

33 Ge Wang and Perry R. Cook. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In *Proceedings of the 2003 International Computer Music Conference, ICMC 2003, Singapore, September 29 - October 4, 2003*. Michigan Publishing, 2003. URL: http://hdl.handle.net/2027/spo.bbp2372.2003.055.

34 Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006. doi:10.1145/1159876.1159877.

35 Gordon Williams. Espruino, 2012. URL: http://www.espruino.com/.

A Appendix A - Scheduling Blinky

Scheduling Blinky

Fig. 14 below shows the timeline of our scheduler executing the blinky program from Listing 5. This chart involves two clocks. The actual wall-clock time, $T_{absolute}$, is represented along the X-axis while the process-local clock, T_{local}, for the process `foo` is shown inside the body of green chart representing `foo`.

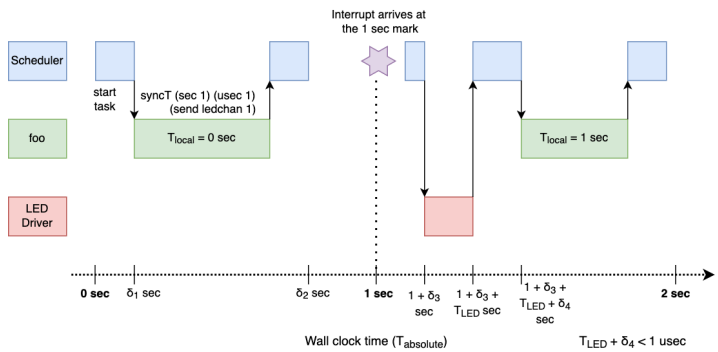


Figure 14 Scheduler timeline while executing the *blinky* program

When the program arrives at the `syncT` statement, an alarm is set for the time at which the VM should begin attempting communication with the LED driver. The alarm is set exactly at the 1-second mark, calculated from the T_{local} clock, which removes the jitter associated with other statement executions and runtime overheads.

Once the alarm interrupt arrives, communication is initiated, and the deadline counter gets activated. The LED driver takes T_{LED} seconds to execute, and the scheduler takes an additional δ_4 time units to unblock the process `foo`. So, the deadline requested to the runtime follows the relation - $\delta_4 + T_{LED} < 1$ *usec*. Finally, T_{local} is incremented by the relation $T_{local} = T_{local} + baseline$, where the baseline is 1 second in our case and the program continues.

B Appendix B - FSM Examples

B.1 Four button blinky

Listing 13 The complete Four-Button-Blinky program (Section 6.1) running on SynchronVM

```
butchan1 = channel ()
butchan2 = channel ()
butchan3 = channel ()
butchan4 = channel ()

ledchan1 = channel ()
ledchan2 = channel ()
ledchan3 = channel ()
ledchan4 = channel ()

press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))

anybutton = choose press1 (choose press2 (choose press3 press4))

program : ()
program =
 let _ = sync anybutton in
 program

main =
  let _ = spawnExternal butchan1 0 in
  let _ = spawnExternal butchan2 1 in
  let _ = spawnExternal butchan3 2 in
  let _ = spawnExternal butchan4 3 in
  let _ = spawnExternal ledchan1 4 in
  let _ = spawnExternal ledchan2 5 in
  let _ = spawnExternal ledchan3 6 in
  let _ = spawnExternal ledchan4 7 in
  program
```

B.2 Large State Machine

Listing 14 The complete complex state machine (Section 6.2) running on SynchronVM

```
butchan1 : Channel Int
butchan1 = channel ()
butchan2 : Channel Int
butchan2 = channel ()
butchan3 : Channel Int
butchan3 = channel ()
butchan4 : Channel Int
butchan4 = channel ()

ledchan1 : Channel Int
ledchan1 = channel ()
ledchan2 : Channel Int
ledchan2 = channel ()
ledchan3 : Channel Int
ledchan3 = channel ()
ledchan4 : Channel Int
ledchan4 = channel ()

not : Int -> Int
not 1 = 0
not 0 = 1

```

```
23 errorLed x = ledchan3
24
25 fail1ev = choose (wrap (recv butchan1) errorLed)
26          (choose (wrap (recv butchan3) errorLed)
27                  (wrap (recv butchan4) errorLed))
28
29 fail2ev = choose (wrap (recv butchan1) errorLed)
30          (choose (wrap (recv butchan2) errorLed)
31                  (wrap (recv butchan3) errorLed))
32
33 led1Handler x =
34    sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)
35
36 led2Handler x =
37    sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)
38
39 led : Int -> ()
40 led state =
41   let fsm1 = wrap (recv butchan1) led1Handler in
42   let fsm2 = wrap (recv butchan3) led2Handler in
43   let ch = sync (choose fsm1 fsm2) in
44   let _ = sync (send ch (not state)) in
45   led (not state)
46
47 main =
48   let _ = spawnExternal butchan1 0 in
49   let _ = spawnExternal butchan2 1 in
50   let _ = spawnExternal butchan3 2 in
51   let _ = spawnExternal butchan4 3 in
52   let _ = spawnExternal ledchan1 4 in
53   let _ = spawnExternal ledchan2 5 in
54   let _ = spawnExternal ledchan3 6 in
55   let _ = spawnExternal ledchan4 7 in
56   led 0
```

C Appendix C - The complete music programming example

We run this program on the STM32F4-discovery board that comes with a 12-bit digital-to-analog converter (DAC), which we connect to a speaker as a peripheral. We can write a value between 0 to 4095 to the DAC driver that gets translated to a voltage between 0 to 3V on the DAC output pin.

To produce a sound note we need to periodically write a sequence of 1's and 0's to the DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is very important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Frequency is related to the periodic rate of a process by the relation:

$$Period = 1/Frequency$$

For instance, the musical note A occurs at a frequency of 440 Hz, which implies it has a time period of 2273 μseconds. From the point of view of the software, we are actually writing two values, a 1 and a 0, so we need to further divide the value by 2 to determine our rate of each individual write. If we call the rate of our writes as $Time_{Write}$, we get the relation -

$$Time_{Write} = Period/2 = 1/(2 * Frequency)$$

Now that we know how to calculate the periodicity of our write in relation to the frequency, we need to know (i) what are the musical notes that occur in the "Twinkle, Twinkle" rhyme and (ii) what are the frequencies corresponding to those notes so that we can calculate the $Time_{Write}$ value from the frequency. The musical notes of the "Twinkle, Twinkle" tune (in the key of G) are well known and is given below:

G G D D E E D C C B B A A G D D C C B B A D D C C B B A

Given the above notes, the frequency of each of these notes are also well known. In Table 2 we show our calculation of the $Time_{Write}$ value for the various musical notes.

Note	Frequency (Hz)	Period (μsec)	Time$_{Write}$ (μsec)
G	196	5102	2551
A	220	4546	2273
B	247	4050	2025
C	261	3822	1911
D	294	3406	1703
E	329	3034	1517

Table 2 Musical notes, their frequencies and time periods

Now we need to specify the time duration of each note. At the end of each note's duration period, we change the frequency of writes to the DAC driver. For instance, consider the transition from the second to the third note of the tune from G to D. If the note duration for G is 500 milliseconds then that implies our writing frequency should be 196 Hz for 500 milliseconds, and then at the 501st millisecond the frequency changes to 294 Hz (D's frequency).

When describing a musical etude, each note should be ideally mapped to its distinct duration in the program. A note duration can be a half note (1000 milliseconds) or a quarter note (500 milliseconds). The note duration of each of the 28 notes of the "Twinkle, Twinkle" tune is given below (Q implies a quarter note and H implies a half note):

Q Q Q Q Q Q H Q Q Q Q Q Q H Q Q Q Q Q Q H Q Q Q Q Q Q H

Listing 15 shows the entire program running on the SynchronVM that cyclically plays the "Twinkle, Twinkle, Little Stars" tune. The first 20 lines consists of declarations initialising a `List` data type and other standard library functions. Lines 72 - 86 consist of the principal logic of the program. Listing 15 can be compiled and run, **unaltered**, on an STM32F4-discovery board.

Listing 15 The *Twinkle, Twinkle* tune (Section 6.3) running on SynchronVM

```
1  data List a where
2    Nil  : List a
3    Cons : a -> List a -> List a
4
5  head : List a -> a
6  head (Cons x xs) = x
7
```

A. Sarkar, B.J. Svensson, M. Sheeran **23:**

```
8  tail : List a -> List a
9  tail Nil = Nil
10 tail (Cons x xs) = xs
11
12 not : Int -> Int
13 not 1 = 0
14 not 0 = 1
15
16 msec : Int -> Int
17 msec t = t * 1000
18
19 usec : Int -> Int
20 usec t = t
21
22 after : Int -> Event a -> a
23 after t ev = syncT t 0 ev
24
25 g : Int
26 g = usec 2551
27
28 a : Int
29 a = usec 2273
30
31 b : Int
32 b = usec 2025
33
34 c : Int
35 c = usec 1911
36
37 d : Int
38 d = usec 1703
39
40 e : Int
41 e = usec 1517
42
43 hn : Int
44 hn = msec 1000
45
46 qn : Int
47 qn = msec 500
48
49 twinkle : List Int
50 twinkle = Cons  g (Cons g (Cons d (Cons d (Cons e (Cons e (Cons d
51           (Cons c (Cons c (Cons b (Cons b (Cons a (Cons a (Cons g
52           (Cons d (Cons d (Cons c (Cons c (Cons b (Cons b (Cons a
53           (Cons d (Cons d (Cons c (Cons c (Cons b (Cons b (Cons a Nil)
54           ))))))))))))))))))))))))))))
55
56 durations : List Int
57 durations = Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons hn
58             (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons
       hn
59             (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons
       hn
60             (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons
       hn Nil)
61             ))))))))))))))))))))))))))))
62
63 dacC : Channel Int
64 dacC = channel ()
65
66 noteC : Channel Int
67 noteC = channel ()
68
69 noteDuration : Int
70 noteDuration = msec 500
```

```
71
72 playerP : List Int -> List Int -> Int -> () -> ()
73 playerP melody nt n void =
74   if (n == 29)
75   then let _ = after (head nt) (send noteC (head twinkle)) in
76        playerP (tail twinkle) durations 2 void
77   else let _ = after (head nt) (send noteC (head melody)) in
78        playerP (tail melody) (tail nt) (n + 1) void
79
80 tuneP : Int -> Int -> () -> ()
81 tuneP timePeriod vol void =
82   let newtp =
83       after timePeriod (choose (recv noteC)
84                                (wrap (send dacC (vol * 4095))
85                                      (λ _ -> timePeriod))) in
86   tuneP newtp (not vol) void
87
88 main =
89   let _ = spawnExternal dacC 0 in
90   let _ = spawn (tuneP (head twinkle) 1) in
91   let _ = spawn (playerP (tail twinkle) durations 2) in
92   ()
```

Our application consists of two software processes and one external hardware process. The $Time_{Write}$ values of each of the fourteen notes are represented as the list `twinkle` on Lines 49-54 and the note durations are contained in the `durations` list (Lines 56-61). We use two channels - `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes.

Owing to the different time periods of the two processes, their T_{local} clock progresses at different rates. In Figure 15 we visualise the message passing that occurs between the two software process and the hardware process when transitioning from a note C4 to a note G4.

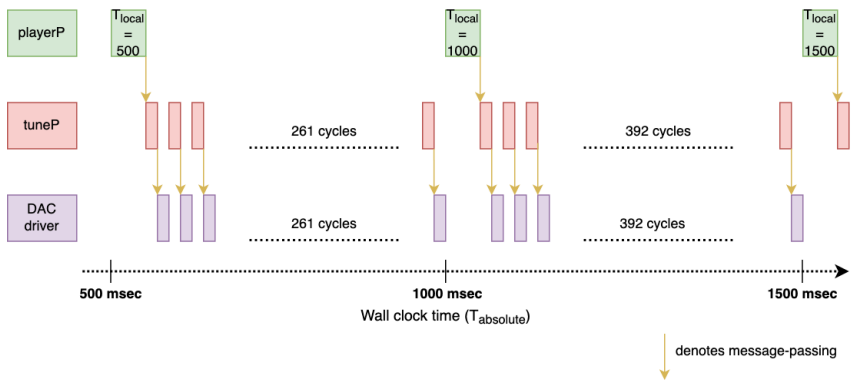


Figure 15 Moving from the note C4 to note G4

As the `playerP` process runs once every 500 milliseconds, the `tuneP` process completes $500 * 10^3/1915 = 261 cycles$ when playing the note C. For the next note, G, the $Time_{Write}$ value changes to 1432 microseconds and the corresponding write frequency changes to 392 cycles and the process cyclically carries on.