Real-time Capabilities in Functional Languages

Jeffrey C Murphy
University at Buffalo
Email: jcmurphy@buffalo.edu

Bhargav Shivkumar
University at Buffalo
Email: bhargavs@buffalo.edu

Lukasz Ziarek
University at Buffalo
Email: lziarek@buffalo.edu

***Abstract*—Functional programming languages play an important role in the development of provably correct software systems. As embedded devices become pervasive and perform critical tasks in our lives, their reliability becomes paramount. This presents a natural opportunity to explore the application of functional programming languages to systems that demand highly predictable behavior. In this paper we explore existing functional programming language compilers and their applicability to real-time, embedded systems.**

I. Introduction

A. Background

Real-time software is an integral part in many systems — for example in power plants, avionics, and medical devices. These systems deliver important services and millions of people rely on them to perform consistently. Writing the software that controls these systems is a difficult undertaking because ensuring correctness and predictability is extremely laborious in the imperative languages that most programmers choose to use.

Functional languages are known for their ability to be reasoned about with respect to correctness, often with automated tools [3, 25, 26]. Given that real-time systems are frequently deployed in safety-critical and mission-critical situations, the ability to write software with a high degree of confidence in its correctness and predictability is important. Functional languages are well suited for the former task (correctness), but are under-represented in the domain of real-time languages and there are relatively fewer research papers associated with investigating how to ensure their real-time predictability.

As discussed in [38], the combination of these two areas are highly desirable as *embedded systems* have become the norm with more than 98% of processors being deployed in such systems, many of which are used in safety-critical applications. Important features of real-time functional programming (RTFP) languages are [61] deterministic behavior, bounded in space/time, asynchronous responsiveness (reactive), concurrent and provable correctness. Hammond [38] additionally identifies periodic scheduling and interrupts/polling models as requirements. Additionally, he observes that a major issue is memory management, especially with respect to unconstrained stack growth – a well known characteristic of recursion-encouraging functional programming languages. We will see how several current functional programming languages support these observations by providing limited or no dynamic memory allocations, no garbage collections and periodic, rather than priority-based, scheduling.

However, with the maturity of language runtime technologies for real-time Java, the availability of a plethora of real-time JVMs [2, 4, 12, 13, 16, 18, 44, 77, 79], which can serve as an execution mechanism for functional languages that compile to bytecode, and the growing interest in leveraging functional reactive programming in real-time systems; researchers are re-examining functional languages and their applicability for real-time systems. Indeed, it is likely due to the rapid advancement of real-time garbage collection [5, 7, 8, 9, 19, 20, 46, 50, 51, 52, 56, 65, 67, 73, 74, 75, 76, 78, 86], scoped memory [10, 28, 41, 69, 70, 96], as well as alternative high-level memory management techniques [1, 6] over the last five years thats has spurred interest in functional languages adapted for real-time. This is due to the fact that most the these results, though developed for real-time Java, are not inherently real-time Java specific. Indeed, functional language developers can leverage the lessons learned in implementing real-time Java runtimes when designing specialized runtime systems for functional languages. Moreover, many languages are now hosted in JVMs and could be ported to more specialized real-time JVMs.

In this paper, we review a variety of functional program languages, with a range of purity and features, and assess each in terms of its suitability for the development and execution of real-time software. We also assess what is lacking in each language and runtime if one were to decide to add a complete real-time feature set to it.

B. Evaluation

For the purposes of evaluating whether a language is capable of supporting real-time features, we will define a prototypical real-time application. The goal of this paper, then, is to evaluate whether or not the application can be written in a given language. The application itself is not intended to exhibit all possible features one would expect in a modern real-time language, but to exhibit a baseline set of features in order to demonstrate whether or not a language has suitable real-time support available. Additionally, this allows us to compare languages from a syntactic point of view and assess how accessible real-time features are to the programmer.

The prototypical real-time application that we will use as a baseline is taken from the Timber Language introduction [88] and is illustrated in Fig. 1

> The following Timber example shows a simple implementation of a sonar driver that is coupled to an alarm. The specifications assumed state that a sonar beep should be 2 milliseconds long, with a maximum jitter of 50 microseconds, and that the required accuracy of the measurements dictate that time-stamps associated with beeps must also be accurate down to the 50 microsecond range. The figure

below illustrates the timing windows constraining the involved methods ping and stop, and thus, indirectly, the actual beep produced. Furthermore, the sonar is supposed to sound every 3 seconds, and the deadline for reacting to off-limit measurements is 5 milliseconds.

In terms of real-time functionality, we intend to evaluate each languages ability to specify periodic scheduling of tasks, start times, dead lines and acceptable variance (jitter). We also will examine the current state of the runtime's garbage collection and its impact on the compiler's ability to deliver a viable real-time application. Finally, we will examine whether or not the compiler is able to produce output that is runnable on an embedded real-time operating system. Table I contains a summary of real-time relevant language features. Source code to the examples presented in this paper is available in our Github archive [64].

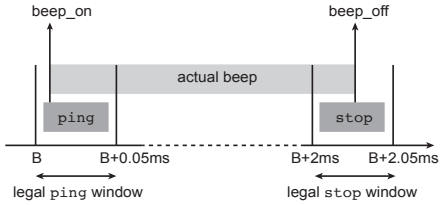


Fig. 1: Timber Sonar example application

C. Definitions

For the purposes of this survey we define a number of terms and metrics that we will leverage in the classification and evaluation of functional languages and their runtimes.

Purity: We define a pure language as one without side-effects, namely purely function without substantial support for I/O. As a rule, we, for the most part, do not consider languages that are pure because, in the context of embedded real-time systems, I/O is a requirement. Semi-pure means the language is functional, preserves referential transparency, but does permit I/O. For example we would classify Standard ML (SML) and Haskell as semi-pure as both languages provide mechanisms for side-effects. We classify languages that support multi-paradigm programming, meaning the language is a mix of functional and imperative constructs (like Scala), as impure. The reason for this classification is that it is up to the programmer to leverage the functional aspects of a multi-paradigm language in order to realize the associated benefits.

Real-time Suitability: We define the real-time suitability of a language based on the current structure of the language runtime (e.g. GC style), linguistic support for real-time constructs (e.g. priorities and tasks), and overall performance and predictability. We say that a language is not suitable for real-time (hard or soft) if the runtime has a stop-the-world Garbage Collector (GC) or other memory management scheme with performance characteristics that are proportional to the size of the area of memory used by the application for dynamic memory allocation (i.e. O(heap)), has no linguistic support for real-time constructs, and / or is interpreted or has an execution model that is fundamentally unpredictable. We classify a language as suitable for soft real-time if it has good performance, a memory management scheme that is concurrent or does not produce pauses on the order of O(heap). Such languages may or may not have explicit linguistic support for real-time constructs. We define suitability for hard real-time based on if a language has *explicit* support for predictable memory management (e.g. RTGC or no runtime allocation), linguistic constructs for specification of priorities and other real-time properties, and evidence of predictable performance.

Language	Purity	RT support	Thread/Task support	Interaction Style [a]	FRP
Atom	Semi	Hard	Other	SM	Supported
Clojure	Semi	Soft	Both	SM	Ready
CoPilot	Semi	Hard	Other	SM	Supported
Erlang	Impure	Soft	Green	MP	Ready
Haskell	Semi	Soft	Other	Both	Ready
Hume	Semi	Hard	Green	MP	Supported
Idris	Semi	Soft	Both	Both	None
Racket	Impure	Soft	Other	MP	Supported
Scala	Impure	Soft	Other	MP	Ready
SML	Semi	Soft	Green	Both	Supported
Timber	Semi	Hard	Native	MP	Ready

[a]SM = Shared Memory, MP = Message Passing

TABLE I: Summary of language attributes

Thread/Task Support: Threads, tasks, or other primitives used to represent a computation are required to express real-time systems that cannot be encoded as a cyclic executive. Coupled with the core mechanism of expressing computation as tasks is the ability to differentiate computations based on priorities as well as express characteristics of the computation (e.g. period and release characteristics). We classify a language's thread or task support depending on the primitives the language provides. We say that the language has *Green* threads if threads are only supported by the runtime library and multiplexed over a single OS thread. We say it supports *Native* threads if the language provides a way to create and manipulate OS threads. A language can also support *Both*, Green and Native threads, or *Other*, which encapsulate a myriad of other concurrency primitives like futures and asynchronous workflows [21].

Interaction Style: Many real-time systems are inherently multi-threaded. This requires interaction between multiple threads of control. We classify a language based on what kind of interaction they allow between threads. The language can support *Shared memory*, where threads modify a common memory location, or *Message Passing* where the threads interact explicitly by sending and receiving messages. A language could also support *Both* styles of interaction. This choice is useful when deciding what language to use for a particular real-time application. For example, Erlang chooses to use only message passing as it is generally used to build large scale distributed systems (soft real-time) and providing controlled access to a shared memory for such a large application would be impractical.

FRP support: With the growing relevance of functional reactive programming (FRP) to real-time systems [49], we mention the extent to which each language supports the FRP style of programming. Languages which support an FRP model deserve a mention as it opens up a possibility of exploiting this model's application to real-time systems. We use the keywords

Supported and *None* to specify if a language has support for FRP or not. We claim that the language supports FRP even it doesn't inherently have an FRP implementation but there are third party libraries that provide it. We also mention that the language is *FRP ready* if it doesn't explicitly support FRP but has constructs, like Actors and message passing interaction, which are the building blocks of an FRP model.

II. Languages not Suitable for Real-Time

As discussed earlier, any programming language requires a few basic features for use in real-time systems. Features like a way to perform I/O, a way to specify priorities of computations, a memory management mechanism (read GC) at the very least are required in order to even start work on making the language real-time friendly. These requirements would obviously rule out most pure functional languages. In this section we take a brief look at such functional languages and comment on why they are not suitable for real-time systems or how much of an effort will be required in order to make them suitable for building real-time systems. Languages that have not had any significant development since long or those that were built for specific research needs are also classified under this section as they seem to be the ones with least support for real-time systems.

A. Charity

Charity is a functional programming language grounded on category theory. Primarily a research language, it lacks IO capability, has no notion of threads or anyway to represent a computation and no concurrency support. Efforts to build a GC supported virtual machine [95] have been attempted but it isn't widespread. There have been no significant updates to the language since 2000 and is very sparingly used for non experimental purposes.

B. Clean

Clean is a general purpose semi-pure functional programming language based on graph rewriting rules and is considerably faster than languages like Haskell due to this. Due to the absence of a threading model or any concurrency support and the lack of a Foreign Function Interface (FFI), we classify it as not suitable for real-time. There have been attempts to implement continuation passing explicitly and build a threading model [93] but it isn't part of the mainstream language development. Presence of uniqueness types could lead to a static GC [90] that might be more deterministic, but there has been no effort towards exploiting this for use in real-time systems.

C. Miranda

Miranda [62] is a semi-pure non-strict functional programming language. It lacks the basic primitives necessary for a real-time system like thread support or predictable memory management. We felt this language deserves a mention since it inspired Haskell, a language more suitable for real-time. It inspired Haskell's lazy evaluation, polymorphic data types, list comprehension among others. The language is currently not under development.

D. Curry

Curry [27] is a functional logic language developed for experimental purposes. Although it has the benefits of a functional language, like lazy evaluation and higher order functions, it also introduces non-deterministic computations that are typical of logical languages. Whether non deterministic functions are suitable for real-time programming is a discussion out of the scope of this paper. Curry supports concurrent evaluation but lacks any "thread like" structures and there is no information on the type of memory management used. There have been some interesting implementations of Curry like the KiCS2 compiler [53] which compiles Curry code to Haskell and a concurrent implementation of Curry in Java [42] which shows more promise to our cause.

III. Languages suitable for real-time

Real-time systems are generally tightly coupled with the physical world, with their utility derived from being able to detect and react to physical events in a predictable amount of time. In order to facilitate the practical development of cyber-physical systems, there are certain language features that we feel are required. Features such as I/O, signals, and state mutation are required in order to construct realistic real-time cyber-physical systems. Other features, such as task scheduling, are highly desirable although not strictly necessary. For example, in the absence of task scheduling, an alternative strategy would be needed such as a cyclic executive [57]. A language with this restriction may be suitable for only a subset of real-time systems. Finally, since real-time systems *react* to the environment they are in, we will examine which languages provide a functional reactive programming model.

In the following sections we examine which languages meet any or all of these goals. We also discuss extensibility and whether it is practical to add the missing features.

A. Haskell

Haskell is a strongly-typed lazy-evaluated purely functional programming language. It is a standardized, actively maintained (n.b. Glasgow Haskell Compiler), language that was designed by a community driven effort dating back to 1990 [48]. I/O is implemented via monads [91] and this approach allows for the preservation of purity which in turn makes programs written in Haskell easier to reason about than imperative languages with I/O [17].

The standard garbage collector included with the Glasgow Haskell Compiler (GHC) is a stop-the-world (STW), generational, copying collector [59]. There are various tunable parameters, for example in-place compaction for the old generation space. However, this GC is not suitable for real-time software since stop-the-world GCs are well understood to have negative impacts on meeting real-time deadlines. For example [73], shows that a STW GC under high load can miss as much as 95% of deadlines.

However, all hope is not lost. Two notable efforts to implement a real-time suitable GC are Microsoft's exploration into multi-core GC with local heaps [60] and a Google Summer of Code (GSoC) project [29] to implement IMMIX for Haskell. The GSoC effort yielded a result that is "not in a state where

it can be included in GHC yet, but it's functional, [doesn't] have known bugs and gets better results than the default GC [under some circumstances]" [30].

The IMMIX GSoC effort shows considerable promise as the IMMIX garbage collector [11] is a hybrid collector that uses a relatively new (in terms of published literature) approach that combines characteristics of mark/sweep and mark/compact into what they term "mark-region". By segmenting memory into chunks, and further segmenting chunks into lines (small consecutive pieces of memory) they are able to optimize marking by propagating marks from the line level to the block level. This allows them to optimize searching for available memory segments. Further, by allocating at the block level they can allow the allocator to be thread-specific and avoid synchronizing at the sub-block (line) level. This improves mutator performance.

They present a novel strategy for defragmentation that is opportunistic – moving lines only when it makes sense, and they combine that with forwarding pointers to further minimize the impact on the mutator. Through careful optimizations and verifying compiler optimizations, they achieve a level of performance that consistently beats the three canonical GC strategies – mark-sweep (MS), semi-space (SS), and mark-compact (MC). Their empirical data covers a wide range of benchmarks and a wide range of metrics including allocation performance, metadata/marking overhead at the line and block level and compaction performance. Another interesting implementation detail was the reservation of blocks (headroom) to facilitate evacuation.

Microsoft's multi-core GC effort involved developing an independent GC capable of running on multiple cores. This allows each processor to perform minor garbage collections independently. They "focus on throughput rather than latency and pause-times" by having processor-local heaps along with a global heap with strictly defined forwarding rules. This is similar to the approach taken by MultiMLton [87] and while it is not immediately applicable to real-time systems, it has promise in that it could potentially be adapted to exploring a per-priority level (rather than per-processor) heap along with an associated per-heap (and therefor per-priority) GC.

In terms of our other significant area of real-time interest, Threads/Tasks, GHC has existing support for concurrency, leading us to conclude that it could be adapted to real-time development with modest effort. One such effort is Atom [45], a domain specific language used in the automotive industry to implement hard real-time functionality. Atom is unusual in that it is a synchronous language, similar to HDL, where its functions are executed based on time (clock cycles elapsed) rather than on events ("reactive programming") or on a priority scheduler. Using Atom, you can construct a schedule that is a set of atomic state transition rules. GHC will validate the schedule and optimize it for you at compile time as opposed to runtime, so that a Real-time OS (RTOS) is not needed to perform this function. The compiler will move rules around and attempt to minimize the Worst Case Execution Time (WCET) of the schedule. The schedule, as mentioned, is based on the periodicity of atomic functions (tasks) that you write. Since tasks are atomic, no locking is needed and this is a key feature needed for Atom's scheduler to correctly calculate WCET. It does task reordering and uses the atomic task definition to avoid having to consider interdependencies during schedule calculation. The recommended scheduling implementation is to use clock interrupts set to trigger a list of tasks designated to run at that interval. Out of the box, Atom will generate a cyclic executive [57] that runs at a predetermined frequency. Tasks execution times are then derived from that frequency.

One limitation to its use in other domains is its lack of dynamic memory allocation (and GC). Atom produces C code that has variables pre-declared with a minimal set of types supported to facilitate low level hardware control and measurement of simple systems. Additionally, and perhaps critically, it does not instrument the scheduler and so does not alert on missed deadlines.

Another promising DSL is Copilot [72], a "language tailored to programming runtime monitors for hard real-time, distributed, reactive systems." Copilot uses Atom to generate hard real-time C code and so generates code that is constant-time and constant-space.

There is also research in the area of RT-FRP [92] with respect to Haskell. Similar to the approach taken by Atom and Copilot, RT-FRP bounds the costs of both execution time and space. The FRP model combines continuous computation with event based computations typical of cyber-physical systems. Applications that combine these two modes of computation, such as robotics, GUIs, medical monitors, and so on are well suited to FRP. The Yale Haskell Group [94] has many examples of embedded Domain Specific Languages (DSL) spanning these, and other, areas that use the FRP model. Since this model is immediately applicable to cyber-physical systems, the work done on RT-FRP is a great starting point for continued research in this area. However, as noted with respect to Atom, etc, research into alternate schedulers, dynamic allocations and GC is needed in order to move from the one-off DSL approach to a more general approach to RT-FRP. There are other Haskell DSLs, such as Dance [47], but we omit covering them for length reasons and since they tend to be similar in implementation and so don't further illuminate the underlying real-time features we wish to explore.

At this time we would categorize GHC, in general, as suitable for soft real-time systems, with promising efforts for future applicability to generalized hard real-time. For existing hard real-time DSLs, some effort would be needed in the areas of GC, dynamic types and scheduler constructs regarding the specification of acceptable jitter so that more intuitive deadline miss detection can be implemented. In general, Haskell's operator overloading and extensibility make it a natural fit for experimentation with novel approaches to applying functional programming concepts in a real-time setting.

B. Erlang

Erlang [33] is a general purpose functional programming language which was developed by Ericsson to build massively scalable soft real-time systems. It was built targeting use in telecommunications domain and has been used widely in that sector. It has a light weight notion of a process, which are not linked to OS threads. This give the language the flexibility of faster context switches and the ability to create many such processes as each process requires as low as 350 bytes initially. In fact, the language creators advertise better performance

over parallel architectures by promoting the creation of more processes.

Since it was built keeping soft real-time applications in mind, Erlang has many real-time features built in. The processes of Erlang already have priorities built in to them and they are even scheduled according to priorities. The general scheduling policy involves separate queues for each of the four specified priority levels and a round robin scheduling within each queue.Processes are pre-empted based on reduction count as opposed to time slices. What Erlang gains in terms of priority based scheduling, it loses in terms of timing constraints. It has little or no support for expressing periodic tasks and resorts to timeouts to express periodicity which do not even provide real-time guarantees. The code snippet in the Table II shows how we can emulate the Sonar example described in the evaluation section using simple timeout structures Erlang provides. Although Erlang has a very exhaustive timer module, use of that is generally a deterrent to performance as it creates a separate process to manage the timers and that can easily get overloaded.

Garbage collection is claimed to be real-time by the developers of the language. Each Process has an individual heap which is managed by a generational copying GC. There is also a global shared Heap between Processes that is entirely managed by Reference Counting garbage collection. Each time the GC occurs inside a process, it pauses only that process which is being GCed and not other Processes. Moreover, the documentation [34] claims that since the heap of each Process is much smaller, the Process is paused for a lesser time. This behaviour has been empirically observed to be suitable for soft real-time applications.

Concurrency is one of the strong points of Erlang and it advocates its strength by using Message passing between processes as its only source of interaction and thus avoiding the disadvantages of shared memory interaction. Since the processes in Erlang are built on top of its Actor model, this throws open an opportunity to explore FRP in Erlang. Values changing over time can be modelled using the Process loops, Signals using the messages passed to Process groups, events as messages to relevant actors and so on. We couldn't find any literature on FRP implemented over Erlang but this is definitely an open area for research.

Although well suited for soft real-time applications, Erlang was built keeping distributed applications in mind. It is noted that fault tolerance of distributed applications, like a messaging interface, is much more relaxed than in applications running on a single machine. There has been a recent project to implement a hard real-time version of Erlang [66], which has support for periodic tasks and a real-time scheduler. But the paper does not propose any changes to the existing Erlang GC. It acknowledges the fact that actual hard real-time cannot be achieved in Erlang until it has a Schism [78] like GC incorporated into it. We conclude by saying that Erlang has all the fundamental building blocks of a real-time functional language but there still are many areas which need to be improved before it can be used for a variety of real-time applications.

C. Hume

Hume [40] is a language based on concurrent autonomous finite state automata (FSA). The focus of Hume is formal analysis, and so we find that its scheduler is cyclic and memory management is focused on static cost space utilization. However, a significant departure from other RTFP languages we reviewed was the use of automatic memory management – Hume uses static analysis to limit space usage. Within the scheduling model, you are able to set timeouts for computation durations.

Hume separates the language into several distinct parts [37], an expression layer that is a purely functional language, a coordination layer (the FSA layer) based on communicating parallel processes, and a declaration layer that supports the other two layers. This allows for targeted proof strategies to be employed that are suited to each layer [39].

D. Racket

Racket [80] is a general purpose multi-paradigm programming language in the LISP/Scheme family. It is advertised as a programming language for the construction of new languages. Racket is extensible through the use of macros [58], which let you specify syntactic extensions to the language. The Racket 3m implementation converts all definitions and expressions to an internal bytecode format and in some platforms this is then Just-in-time (JIT) compiled to native code. Although there is no real-time implementation of the language, it is definitely worth looking at properties that are conducive to making it suitable for real-time systems.

Racket has a really good threading library with support for green threads as well as OS threads. Each thread when created is assigned to a Custodian which is responsible for managing all threads (and TCP ports, file stream ports etc). Expressing priorities in these threads maybe possible by either changing the definition of a thread in the runtime system to support a notion of priorities or by grouping threads of similar priority under the same sub custodian. We could also create a direct OS thread from Racket using Places [82] and then allow the RTOS to schedule them according to its scheduling policies. Racket also supports true parallelism using futures [81].

The 3m implementation of Racket has a stop-the-world, generational GC that supports incremental collection, which currently might not be suitable for real-time purposes. However, the language supports the re-use of storage when the GC can prove the object to be unreachable. This support is provided through the use of weak references, which the GC maintains for reuse. FrTime [35] is a functional reactive extension of Racket. Like any other FRP system, FrTime treats state as a time varying value specifically called behavior.

Being a multi-programming paradigm, introduction of non functional code makes a Racket program hard to reason about and it is due to this that we classify Racket as being impure according to definitions in section I-C. Although Racket has some suitable basic properties needed for supporting real-time systems, there has been no attempt to modify the language towards supporting them. There needs to be many changes like a real-time GC, structures to specify periodicity and priority of tasks, deadline aware scheduling of tasks among others before it can be used to build real-time applications.

E. Scala

Scala is a JVM based language where functional constructs can co-exist with Java code [68]. Scala is designed to be concise while assuring the programmer type-safety. Additionally, it can interact with code written in Java, opening up the possibility of using the Real-time Specification of Java (RTSJ) [14] and leveraging a real-time JVM. Since Scala does not diverge too much from Java, it is an approachable functional language for programmers.

Scala concurrency is based on actors and built on top of Java's threading model. The default GC for Scala is that of the underlying JVM – i.e. throughput oriented, but without specific guarantees on the duration of pause times. There has been recent interest in exploring Scala's suitability in a real-time setting [84]. The features of Scala that appeal to the real-time Java community are scalability, that the language is statically typed and easy to use with Java libraries, and the observation that Scala's functional nature matches the RTSJ's scoped memory model and also helps real-time garbage collection by eliminating use of shared state. The possibility of building a real-time DSL as a Scala library, which would delegate to the RTSJ for implementation, is envisioned as future work.

Scala's support for Actor based concurrency model and message passing allow for easily constructing FRP models. This makes Scala particularly interesting as a mechanism for real-time FRP systems research. Although the language does not have an inbuilt FRP, there are third party libraries [43] available that provide basic FRP support. We classify Scala to be useful for soft real-time as it has the feature set necessary to build real-time systems and a throughput oriented GC.

F. SML

Standard ML (SML) is a general purpose, strongly typed, functional programming language. Typically SML based languages are eagerly evaluated. There are two efforts we wish to discuss with respect to real-time ML: ML-Kit [32] and RTMLton [55]. Additionally, we will discuss two derivatives of ML: F# and Ocaml.

ML-Kit runs on Linux/x86 and uses a region-based memory management with a reference-tracing GC. Region-based memory management allows for the efficient coincidental deallocation of related memory segments, which in turn lessens the amount of work the GC needs to perform on general deallocation. This is important because it improves the programmer's ability to estimate the worst-case performance of the GC for a particular application. [36] demonstrates that region-based management does, in fact, dramatically reduce the number of GCs required. However, it also increases the amount of overall space that is utilized, due to intra-region wasted space. Since GC is one of the most difficult part of reasoning about real-time execution (most functional languages that support real-time typically omit GC and opt for static allocation) it stands to reason that an allocation system that minimizes the frequency and duration of GC pauses is a positive feature for an language runtime. The use of regions is not invisible to the programmer, though, and some effort must be invested [89] in proper application design and profiling to reap the maximum benefits of this technique. ML-Kit lacks tasks or threads and so priority-based, real-time concurrency is not possible without investing in those areas. Additionally, ML-Kit has limited FFI (foreign function interface) capabilities, so driving ML functions from OS generated interrupts is not possible. This means that an investment in that area is also needed in order to implement a timing based cyclic executive as we saw in Atom and Timber. Some preliminary, unpublished, work [71] has been done to investigate improving ML-Kit in these areas.

RTMLton [55], a derivative of MLton [63], reorganizes MLton's memory allocation scheme so that it uses uniformly sized chunks of memory. By analyzing allocation sizes, an optimal chunk size can be derived to yield constant time allocations. Additionally, by dividing memory into uniform chunks, no compaction is required during GC, allowing for a reduction in the time complexity estimation of a GC run. MLton includes multiple ways of concurrency abstraction, and so RTMLton has been able to build on that work to provide preliminary work in the area of priority-based real-time scheduling and support for RTEMS (an RTOS that runs on a variety of hardware). It does not support deadline-based scheduling or a cyclic executive. It does, however, have a robust FFI, allowing for rapid integration with an RTOS that has a C API and so an interrupt driven deadline-based scheduler is feasible. The flexibility of MLton's runtime holds promise for experimenting with alternate scheduling and programming models.

F# belongs to the ML family and is implemented on .NET, it provides asynchronous workflows via tasks which allows for asynchronous composability [97]. This is interesting because it allows for experimentation with new theoretical scheduling models. We were unable to find evidence of any work being done to extend F# with real-time capabilities.

OCaml uses a mark-sweep GC combined with separate short-lived and long-lived object heaps. Allocations are dynamic and so periodic compaction is required. These GC features imply that an overhaul of the GC would be required to make it suitable as an RTFP language. OCaml includes a robust FFI and so integration with an RTOS is feasible, opening the door to priority-based and deadline-based scheduling at a minimum. For concurrency, OCaml combines tasks with a reactive model in what is called the Async library. This library is built in the concept of `Deferred`s (`Futures`) and `Monadic` constructs in order to provide asynchronous operations. From a runtime perspective, OCaml lacks both a deterministic order of evaluation and an equality function that is guaranteed to terminate [54] which may imply that its execution model is fundamentally unpredictable and so should not be considered as an RT candidate language. More research is needed to determine whether or not this true.

G. Clojure

Clojure is a JVM based dialect of Lisp with an emphasis on functional programming. Although primarily hosted on the JVM, implementations of Clojure exists for the Common Language Runtime [22] and Javascript [24]. Unlike Scala, it does not allow mixing of Java code with Clojure code and thus is purely functional. It encourages immutability and it acknowledges mutable state through a concept of Identities [23] i.e. series of immutable states over time. Clojure allows calls into Java using the "." target member notation. This opens up a

plethora of avenues as it can exploit the massively exhaustive Java Libraries. Clojure uses the Java threading system and leverages software transactional memory [85] for concurrency control. Clojure also has third party libraries that provide an FRP implementation. Because Clojure is hosted on a JVM, we consider it suitable for soft real-time use. Clojure compiles to Java bytecode and as such leverages the JVM's GC, threads, and other runtime constructs. Much like other languages host in the JVM, there is a possibility of using a real-time VM [79] for Clojure.

H. Idris

Idris [15] is a dependently typed, semi-pure (supports IO), functional programming language that is eagerly evaluated but with support for optional lazy evaluation. It is primarily used for interactive theorem proving. Idris has C, Java, and JavaScript backends and also is robust enough to support addition of new backends without affecting the rest of the compiler. There is no real-time version of the language that has been implemented. It makes use of the target backend's threading support, through FFI calls, to provide a minimal message passing concurrency. There are projects that provide other backends, like Erlang [31], to provide better concurrency support. Idris' runtime uses a cheney-copy GC, which is not real-time aware and not performant enough for even soft real-time applications. Idris, however, is an interesting candidate for functional real-time systems as apart from its ability to leverage backend language constructs, it supports uniqueness types, which may reduce the need for GC through static memory management techniques, and dependent types, which allow for verification of program properties through static type checking. We classify Idris as suitable for soft real-time as it has prospects for being modified towards use in building soft real-time systems.

I. Timber

Timber is a functional programming language targeted at embedded real-time systems. It includes concurrency, strong timing constraints that influence the output of its scheduler, event-driven reactions, and objected oriented modeling. Development of the language started in 2000 as part of the DARPA Program Composition for Embedded Systems (PCES) program.

Timber includes a mini POSIX based RTOS with threading, a GC and a cyclic executive. The GC allows dynamic allocations with support for basic types in addition to arrays and tuples, but is not slack-based and instead executes when heap utilization exceeds a certain threshold. This is not ideal for hard real-time applications and an interesting area of future research would be alternative GC strategies such as region-based [83] allocations. A review of the GC code comments shows at least one edge case that is potentially unhandled – objects that contain mutable arrays. The included scheduler is based on explicit timing constraints entered by the programmer, including, for example, start times, deadlines and future actions. This information allows the compiler to compute predictable timing constraints using an EDF strategy when scheduling tasks as well as detect schedule misses. Timber event handling mechanisms are able to deal with time based interrupts, file (including standard input) and (TCP only) network I/O events.

The Timber compiler is written in Haskell, but unlike Atom (discussed in III-A), the language itself is not Haskell compatible. The compiler is able to emit C code that can be cross compiled onto an RTOS, allowing for the development of hard real-time applications, however development of the compiler, based on postings to the Timber mailing list, seems to have stopped around 2009-2010 and we were unable to trivially get the compiler to work on a contemporary release of GHC.

IV. Related Work

A. Other Languages

There are many interesting languages that, due to space considerations, we are unable to cover. Some brief words on just a few of them:

- *Elm* is an FRP, synchronous, language that targets the web browser as a runtime environment. This environment is unsuitable for a real-time setting.
- *Frege* is a Haskell like language that runs on the JVM. Interestingly it compiles frege source to Java source code and not bytecode. We can classify it as being capable of soft real-time support since it has the JVM support and can natively use Java libraries.
- *Mercury* is a functional logic language with many backends including C, Java and Erlang. No current real-time implementation but it has potential for exploration due to being able to compile to and call functions of the backend language.

B. Real Time Garbage Collection (RTGC)

There are roughly three classes of RTGC: (i) time based [7] where the GC is scheduled as a task in the system, (ii) slack based [78] where the GC is the lowest priority real-time task and executes in the times between release of higher priority tasks, and (iii) work based [86] where each allocation triggers an amount of GC work proportional to the allocation request. In each of these RTGC definitions, the overall system designer must take into consideration the time requirements to run the RTGC.

V. Conclusion

In this paper we've seen a variety of functional programming languages with a number of real-time promoting attributes. Some of those attributes, such as I/O and task modelling, are requirements for producing useful real-time applications while other attributes, such as dynamic memory management, are desirable but not mandatory. Languages with extensive concurrency support are more suitable for adaptation to real-time systems since they provide more opportunities to adapt their programming models to the underlying model of the RTOS you are targeting. There are a languages, like Haskell, that, by virtue of the flexibility of their compiler, are host to a broad ecosystem of real-time DSLs. Other languages are more purpose-built, like Hume, and as a consequence have more depth in terms of proving and analysis infrastructure. Finally, there are languages like Scala that have not yet seen significant investments in real-time at the linguistic level, but, due to their use of the JVM, have substantial promise in the area of real-time research.

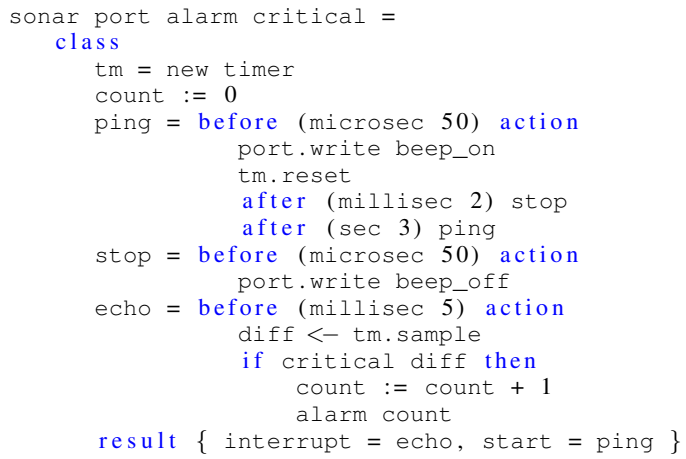
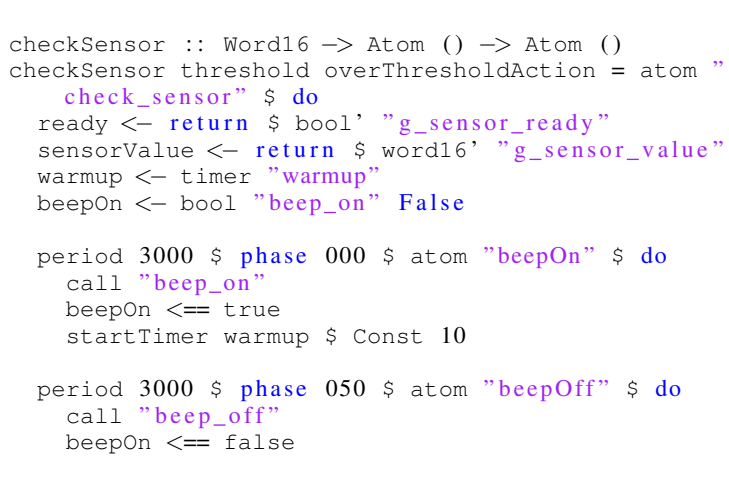
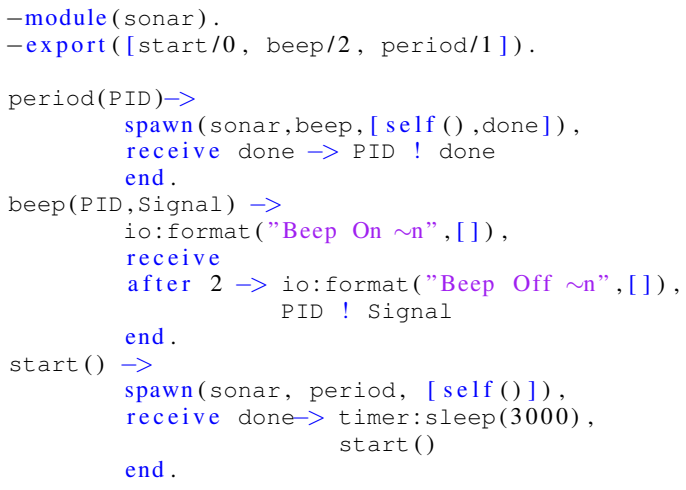
Language	Code Example	Notes
Timber		Canonical example. Interesting points: • the ability to specify actions to take place if deadlines are missed. For example, "before (microsecond 50)" indicates that the routine must execute before that amount of time has elapsed after it starts. • the recursive scheduling of "ping" that includes the frequency.
Atom (Haskell)		Notes: • Functions are scheduled by period, with an optional offset ("phase") • Specifying acceptable jitter is not intrinsic to the semantics, possibly requiring a monitor routine to fire frequently in order to check that that deadlines have been met.
Erlang		Notes: • Doesn't support explicit periodic scheduling so the timers aren't guaranteed to fire off at correct time. • The use of timer:sleep() does not affect performance as it does not create a separate process

TABLE II: RT Examples For Select Languages

REFERENCES

[1] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Real-Time Syst.*, 37(1):1–44, Oct. 2007.

[2] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7(1):5:1–5:49, Dec. 2007.

[3] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568 – 589, 2009. Special Issue on Mathematics of Program Construction (MPC 2006).

[4] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE Int'l Conf. on Embedded software*, EMSOFT '07, pages 249–258, New York, NY, USA, 2007. ACM.

[5] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM Int'l Conf. on Embedded Software*, EMSOFT '08, pages 245–254, New York, NY, USA, 2008. ACM.

[6] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for Java. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conf. on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 1–11, New York, NY, USA, 2008. ACM.

[7] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *Proceedings of the 2003 ACM SIGPLAN Conf. on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 81–92, New York, NY, USA, 2003. ACM.

[8] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[9] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, Mar. 1992.

[10] W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time Java. In *Proceedings of the First Int'l Workshop on Embedded Software*, EMSOFT '01, pages 289–305, London, UK, UK, 2001. Springer-Verlag.

[11] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, June 2008.

[12] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A predictable Java profile: rationale and implementations. In *Proceedings of the 7th Int'l Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 150–159, New York, NY, USA, 2009. ACM.

[13] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making hotspot" real-time. In *Proceedings of the Eighth IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[14] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[15] E. BRADY. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[16] E. J. Bruno and G. Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[17] A. Butterfield and G. Strong. *Implementation of Functional Languages: 13th Int'l Workshop, IFL 2001 Stockholm, Sweden, September 24–26, 2001 Selected Papers*, chapter Proving Correctness of Programs with IO —A Paradigm Comparison, pages 72–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[18] A. Cavalcanti, A. Wellings, J. Woodcock, K. Wei, and F. Zeyda. Safety-critical Java in circus. In *Proceedings of the 9th Int'l Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 20–29, New York, NY, USA, 2011. ACM.

[19] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM.

[20] H. Cho, C. Na, B. Ravindran, and E. D. Jensen. On scheduling garbage collector in dynamic real-time systems with statistical timing assurances. *Real-Time Syst.*, 36(1-2):23–46, July 2007.

[21] A. Cisternino, A. Granicz, and D. Syme. *Expert F# 4.0*. Apress, 2015.

[22] Clojure implemented on clr. https://github.com/clojure/clojure-clr.

[23] State as identities. http://clojure.org/about/state.

[24] Clojure implemented on javascript. https://github.com/clojure/clojurescript.

[25] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, pages 55–59, New York, NY, USA, 2007. ACM.

[26] E. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3pat, an approach for certified automated termination proofs. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 63–72, New York, NY, USA, 2010. ACM.

[27] Curry programming language official website. http://www-ps.informatik.uni-kiel.de/currywiki/.

[28] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time Java. In *Proceedings of the 3rd Int'l symposium on Memory management*, ISMM '02, pages 132–142, New York, NY, USA, 2002. ACM.

[29] M. T. G. e Silva. Implementing the immix garbage collection algorithm on ghc. https://www.google-melange.com/gsoc/project/details/google/gsoc2010/marcot/5639274879778816, 2010.

[30] M. T. G. e Silva. Porting immix to haskell as a gsoc project. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC/Immix, 2010.

[31] A. S. Elliott. A concurrency system for idris and erlang. http://lenary.co.uk/publications/dissertation/Elliott_BSc_Dissertation.pdf, 2015.

[32] M. Elsman. Mlkit documentation. http://www.elsman.com/mlkit/.

[33] Erlang programming language official website. http://www.erlang.org/.

[34] Garbage collection in erlang. http://erlang.org/faq/academic.html#idp33101392.

[35] FRP in racket programming language. http://docs.racket-lang.org/frtime/.

[36] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. *SIGPLAN Not.*, 37(5):141–152, May 2002.

[37] K. Hammond. *Implementation of Functional Languages: 12th Int'l Workshop, IFL 2000 Aachen, Germany, September 4–7, 2000 Selected Papers*, chapter The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour, pages 122–139. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[38] K. Hammond. Is it time for real-time functional programming? In S. Gilmore, editor, *Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, United Kingdom, 11-12 September 2003.*, volume 4 of *Trends in Functional Programming*, pages 1–18. Intellect, 2003.

[39] K. Hammond, C. Ferdinand, and R. Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Rev.*, 3(4):27–36, Oct. 2006.

[40] K. Hammond, G. Michaelson, and R. Pointon. The hume report, version 1.1. http://www-fp.cs.st-andrews.ac.uk/hume/report/hume-report.pdf.

[41] H. Hamza and S. Counsell. Region-based RTSJ memory management: State of the art. *Sci. Comput. Program.*, 77(5):644–659, May 2012.

[42] M. Hanus and R. Sadre. A concurrent implementation of curry in java. In *In Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[43] L. Haoyi. Scala.rx,frp implementation in scala. https://github.com/lihaoyi/scala.rx.

[44] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the JavaTM virtual machine. In *Proceedings of the Fourth Int'l Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '01, pages 53–, Washington, DC, USA, 2001. IEEE Computer Society.

[45] T. Hawkins. Atom: A synchronous hard real-time edsl for ghc. https://github.com/tomahawkins/atom.

[46] M. T. Higuera-Toledano and V. Issarny. Improving the memory management performance of RTSJ: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):715–737, Apr. 2005.

[47] L. Huang and P. Hudak. Dance: A declarative language for the control of humanoid robots. Technical Report YALEU/DCS/RR-1253, Yale University, August 2003.

[48] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conf. on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[49] R. Kaiabachev, W. Taha, and A. Zhu. E-frp with priorities. In *Proceedings of the 7th ACM & IEEE Int'l Conf. on Embedded software*, pages 221–230. ACM, 2007.

[50] T. Kalibera. Replicating real-time garbage collector for Java. In *Proceedings of the 7th Int'l Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 100–109, New York, NY, USA, 2009. ACM.

[51] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst.*, 29(3):8:1–8:29, Aug. 2011.

[52] S. Kato, Y. Ishikawa, and R. R. Rajkumar. Cpu scheduling and memory management for interactive real-time applications. *Real-Time Syst.*, 47(5):454–488, Sept. 2011.

[53] Kics2: A curry compiler with haskell target. http://www-ps.informatik.uni-kiel.de/kics2/.

[54] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, Jan. 2014.

[55] M. Li, D. E. McArdle, J. C. Murphy, B. Shivkumar, and L. Ziarek. Adding real-time capabilities to a sml compiler. In *DPRTCPS: The First Workshop on Declarative Programming for Real-Time and Cyber-Physical Systems*, DPRTCPS '15, pages 1–6, New York, NY, USA, 2015. ACM SIGBED.

[56] T. F. Lim, P. Pardyak, and B. N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the 1st Int'l symposium on Memory management*, ISMM '98, pages 118–129, New York, NY, USA, 1998. ACM.

[57] C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53.

[58] Macros in racket. https://docs.racket-lang.org/guide/macros.html#%28tech._macro%29.

[59] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th Int'l Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[60] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the Int'l Symposium on Memory Management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM.

[61] J. A. McDermid. Computing tomorrow. chapter Engineering Safety-critical Systems, pages 217–245. Cambridge University Press, New York, NY, USA, 1996.

[62] Miranda official website. http://miranda.org.uk/.

[63] MLton. http://www.mlton.org.

[64] J. C. Murphy, B. Shivkumar, and L. Ziarek. https://github.com/UBMLtonGroup/survey_sonar, 2016.

[65] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, PLDI '93, pages 217–226, New York, NY, USA, 1993. ACM.

[66] V. Nicosia. Towards hard real-time erlang. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07, pages 29–36, New York, NY, USA, 2007. ACM.

[67] K. Nilsen. High-level dynamic memory management for object-oriented real-time systems. *SIGPLAN OOPS Mess.*, 7(1):86–93, Jan. 1996.

[68] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[69] oSCJ. Computer science department annual report, Purdue University. http://www.ovmj.net/oscj/publications/Documents/oSCJ-Report.pdf, Jan. 2010.

[70] F. Parain. Region-based memory management for real-time Java. In *Proceedings of the Fourth Int'l Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '01, pages 387–, Washington, DC, USA, 2001. IEEE Computer Society.

[71] J. Penner. Undergraduate honours project: Provably safe real-time programming. *Unpublished.*, 2004.

[72] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conf. on Runtime Verification*, LNCS. Springer, November 2010. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/rv2010.html.

[73] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: A real-time garbage collector for multiprocessors. In *Proceedings of the 6th Int'l Symposium on Memory Management*, ISMM '07, pages 159–172, New York, NY, USA, 2007. ACM.

[74] F. Pizlo, A. L. Hosking, and J. Vitek. Hierarchical real-time garbage collection. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 123–133, New York, NY, USA, 2007. ACM.

[75] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM.

[76] F. Pizlo and J. Vitek. Memory management for real-time java: State of the art. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 248–254, Washington, DC, USA, 2008. IEEE Computer Society.

[77] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European Conf. on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

[78] 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 Conf. on Programming language design and implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[79] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the Int'l Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, 2009.

[80] Racket programming language. https://www.racket-lang.org/.

[81] Futures in racket programming language. http://docs.racket-lang.org/reference/futures.html.

[82] Places in racket programming language. https://docs.racket-lang.org/reference/places.html?q=place#%28tech._place%29.

[83] G. Salagnac, C. Nakhli, C. Rippert, and S. Yovine. Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, in association with the 20th ACM ECOOP Conf.*, Nantes, France, July 2006.

[84] M. Schoeberl. Scala for real-time systems? In *Proceedings of the 13th Int'l Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, pages 13:1–13:5, New York, NY, USA, 2015. ACM.

[85] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[86] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th Int'l Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM.

[87] K. C. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming*, 24:613–674, 2014.

[88] Timber: A gentle introduction. http://www.timber-lang.org/index_gentle.html.

[89] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, January 2006.

[90] Basic ideas behind uniqueness typing. http://clean.cs.ru.nl/download/html_report/CleanRep.2.2_11.htm#_Toc311798093.

[91] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[92] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *Proceedings of the Sixth ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP '01, pages 146–156, New York, NY, USA, 2001. ACM.

[93] A. V. Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In *The 14th Int'l Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers, volume 2670 of LNCS*, pages 215–231. Springer, 2002.

[94] Yale haskell group. http://haskell.cs.yale.edu/.

[95] M. Zeng. An implementation of charity. Master's thesis, The University of Calgary, 2003.

[96] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE Int'l Real-Time Systems Symposium*, RTSS '04, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.

[97] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Composable asynchronous events. *SIGPLAN Not.*, 46(6):628–639, June 2011.