Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications

Martin Schoeberl
Technical University of Denmark, Lyngby, Denmark

Ehsan Khodadad
Technical University of Denmark, Lyngby, Denmark

Shaokai Lin
University of California, Berkeley, CA, USA

Emad Jacob Maroun
Technical University of Denmark, Lyngby, Denmark

Luca Pezzarossa
Technical University of Denmark, Lyngby, Denmark

Edward A. Lee
University of California, Berkeley, CA, USA

Abstract

Real-time systems need to prove that all deadlines will be met. To enable this proof, the full stack of the system must be analyzable, and the right tools must be available. This includes the processor (execution platform), the runtime system, the compiler, and the WCET analysis tool.

This paper presents a combination of the time-predictable processor Patmos, the coordination language Lingua Franca, and the WCET analysis tool PLATIN. We show how carefully written Lingua Franca programs enable static WCET analysis to build safety-critical applications.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Automated static analysis

Keywords and phrases worst-case execution time, coordination language, real-time systems, lingua franca

Digital Object Identifier 10.4230/OASIcs.WCET.2024.4

Supplementary Material
Software (Lingua Franca Source Code): `https://github.com/lf-lang/`
Software (T-CREST Source Code): `https://github.com/t-crest/`
Software (Experiment Instructions): `https://github.com/lf-lang/lf-patmos-template/`
archived at `swh:1:dir:65a90fb54c7c1753a6fc062f1e19c41c0ca5dadd`

Funding *Emad Jacob Maroun*: Huawei Technologies Sweden AB, Agreement Number TC20220819029.

1 Introduction

In safety-critical applications, the reliability of a system is of utmost importance to ensure no catastrophic failures happen. Real-time systems must guarantee that they respond to events within a given deadline. Designing such a system includes analyzing all its tasks and ensuring its worst-case execution time (WCET) and that the resulting schedule will meet its deadlines. However, it is not enough that individual tasks meet their deadlines. We must also ensure that all the tasks always meet their deadlines, regardless of the runtime conditions or other tasks executing in parallel. As such, schedulability analysis must consider not only the individual tasks but also their runtime environment and execution platform.

22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024).
Editor: Thomas Carle; Article No. 4; pp. 4:1–4:13
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A reliable real-time system must be built from a stack of analyzable components. It starts with a platform based on a time-predictable processor. Then comes the execution environment, which includes a real-time operating system or real-time runtime environment that can allocate resources as needed. Next comes programming language and compiler that must support the writing of analyzable code. Lastly comes the analysis tool, which should be able to account for all the components to produce safe WCET bounds, i.e., bounds that are guaranteed not to be lower than the actual WCET.

This paper presents a complete real-time system and toolchain for safety-critical applications. It is based on the time-predictable RISC processor Patmos [33]. Instead of a complicated real-time operating system, we use the Lingua Franca (LF) reactor language as our runtime environment [24]. LF allows us to write individual *reactions* in the C language, which are automatically compiled with the LF runtime using the Patmos compiler. The LF runtime handles the provisioning of resources to each reaction and schedules them when needed. Using the Platin WCET analysis tool, we can analyze the WCET of individual reactions and feed it to a quasi-static schedule generator that ensures reactions are scheduled such that they will never miss a deadline. The quasi-static schedule generator and its underlying virtual machine, PretVM, have recently been introduced to LF to enable fine-grained timing analysis of the LF runtime.

The individual components presented in this paper are not new; they have been presented in other papers. However, this paper's contribution is the presentation of the combination of those components to provide a complete time-predictable execution environment. This is intended as a step towards building correct-by-construction real-time systems.

Lingua Franca, Patmos, and Platin are open-source. The links to the GitHub repositories are given on the title page. To reproduce the evaluation, consult the `README` file in the following repository: `https://github.com/lf-lang/lf-patmos-template/`.

The rest of this paper is organized into the following sections. Section 2 provides the background on Lingua Franca, the Patmos time-predictable processor, the compilation pipeline, and the Platin WCET analysis tool. Section 3 discusses the most important aspects of analyzable code in LF applications, as well as the benefits and issues of LF for WCET analysis. Section 4 presents the experimental evaluation results of the proposed approach. Section 5 discusses related work. Section 6 concludes the paper.

2 Background

This paper combines several technologies to build a complete real-time platform: the coordination language Lingua Franca [22], the Patmos processor [33] and compiler, and the WCET analysis tool Platin [25].

2.1 Lingua Franca

LF is a coordination language and framework based on concurrent actors called Reactors. LF adds deterministic reactive concurrency to target languages. Currently, it supports C, C++, Python, TypeScript, and Rust. The generated code can be deployed on almost every computer system, including embedded systems [5, 22, 23].

Coordination languages and frameworks are designed based on coordination models of computations. These models of computation provide a technology that supports the interaction between software components. Moreover, they generally enhance modularity, reuse of existing (sequential or even parallel) components, portability, and language interoperability [27, 34].

In the coordination languages and frameworks, actors are usually used as the primary programming model. This programming model was first introduced in 1973 by Hewitt for concurrent systems. They are independent entities like objects that can communicate through asynchronous message passing without any locking mechanism [11].

To make actors deterministic, a new model, named *reactors*, was introduced in 2019 as the building block of the Lingua Franca language. In reactors, messages are guaranteed to be delivered to a reactive component in order. For these purposes, logical timestamps are used [24].

Procedures inside reactors are called reactions invoked in response to a trigger event. The reactions are atomic to one another, meaning they are mutually exclusive. The reactions can be written in the LF's target programming languages, and they can read input and produce outputs. Timers, ports, actions, and built-in triggers (such as startup or shutdown) can trigger reactions.

Ports are the types inside the reactors responsible for communicating with other reactors. We have two kinds of ports, input, and output, for receiving and sending messages. LF uses a timestamp tag on a logical timeline for messages to make them ordered. Unlike physical time, logical time does not elapse during reaction execution. In LF, timers use logical time to invoke reactions periodically.

2.2 Patmos

Patmos [33] is a RISC-style processor developed as part of the T-CREST project [32]. Patmos is designed for real-time systems with ease of analysis in mind. It has features that make it easy to analyze, such as an in-order pipeline and special caches. Instead of a traditional instruction cache, Patmos includes a method cache that stores complete function bodies or explicit parts of functions (sub-functions) [6]. At function calls/returns or at explicit points in the function, the method cache is triggered to load the next executed (sub-)function. This means instruction fetching can only miss in the method cache at this point, making it easy for an analyzer to reason about. Patmos also includes a stack cache that stores stack-local data exclusively and is explicitly controlled by the compiler [15]. Accessing stack data, therefore, never misses except at the start or end of a function. The remaining data accesses go through the conventional data cache or can circumvent all caching to access main memory directly.

2.3 The Compiler and the WCET Analyzer

In this paper, we use C as the target language and describe the compilation pipeline of LF for that target. LF, as shown in Listings 1 and 2, contains target code in C, wrapped with the markers `{=` and `=}`. The code around those C fragments is written in LF, which the LF compiler compiles into C functions. The generated functions include those C fragments. Additionally, LF provides the runtime (e.g., the reactor scheduler, functions to set outputs, and other utility functions) as C source files. The LF library code also contains platform-specific low-level functions. LF can execute on various platforms, from systems with full-blown operating systems (e.g., Linux or MacOS) down to the bare metal. In the latter case, LF *is* the operating system, and since no complex operations are used, the timing of the full application can be analyzed.

The generated C code is then compiled with a C compiler into an executable. This paper uses Patmos as the execution platform, which has had LF ported to it [16]. The Patmos compiler is based on the LLVM framework, which compiles C language code to Patmos machine code. It supports adding flow fact information to the code using annotations, the

Listing 1 The `Source` reactor of `SimpleConnection`.

```
reactor Source {
  output out: int
  timer t(0, 1 sec)
  state s: int = 0

  reaction(t) -> out {=
     lf_set(out, self->s);
     self->s++;
  =}

  reaction(t) -> out {=
    int v = -1 * self->s;
    lf_set(out, v);
  =}
}
```

Listing 2 The `Sink` reactor (incomplete) and reactor connections of `SimpleConnection`.

```
reactor Sink {
  input in: int
  state last_received: int = 0

  reaction(in) {=
    self->last_received = in->value;
  =}
  ...
}

main reactor {
  source = new Source()
  sink = new Sink()
  source.out -> sink.in after 2 sec
}
```

simplest of which are loop-bound annotations. The flow facts are maintained through the compilation pipeline and can be exported as part of the compilation to be used by the Platin WCET analyzer. The compiler automatically inserts code to manage the Patmos method and stack caches so that programs experience misses only at the predefined points.

We use the open-source analysis tool Platin to derive WCET bounds for code executing on Patmos [10, 25]. Through the tight integration with Patmos' compiler, Platin gets details about the program control flow and flow facts, which it uses to estimate the WCET bound. Platin includes a detailed model of the Patmos processor architecture, its method cache, and its stack cache [13, 15]. It does not have a dedicated data-cache analysis, meaning all data-cache accesses are assumed to be cache misses.

A requirement for using Platin is that the code is analyzable and the compiler can provide it with a *.pml* file containing control flow and flow fact information. As such, the LF runtime and scheduling code must be analyzable and include sufficient information and annotations for Platin to estimate the WCET of the LF runtime functions.

3 Analysis-Friendly Applications with Lingua Franca

The execution time of general programs is usually not statically analyzable. We need restrictions in the algorithms, e.g., maximum bounds on loop iterations, and a runtime system amenable to timing analysis, e.g., Lingua Franca, thanks to its determinism.

3.1 Analyzable Code

Code for real-time systems must be carefully written to enable static analysis tools, like Platin, to determine upper bounds on execution times, the WCET bounds [28]. The most apparent restriction on real-time code is the prohibition of indeterminate loop iteration. As such, annotations must be added to the code to provide such a bound when the tools or compilers cannot infer an upper bound on the number of iterations a loop may execute. Likewise, recursion is often prohibited in real-time code, as it also introduces the possibility of indeterminable execution time through infinite recursion. Recursion depth bounds can also be used to limit recursion. However, Platin does not support analysis of recursion, meaning we also prohibit recursion.

Dynamism must also be strictly regulated in real-time code. Dynamically sized arrays or data structures must have an upper bound on their size, such that iteration over those arrays is also bounded. Dynamic function pointers are also often prohibited, as knowing which functions they call is difficult. Platin does not allow any function calls through function pointers.

Furthermore, the C standard library contains several functions that are not analyzable. One prominent one is `printf`. Besides being a complex function with probably unbounded loops, `printf` may block. In the case of Patmos, the standard output stream is mapped to a serial port. When the send buffer of the serial port is full, `printf` will block until characters are sent out.

Another functionality to avoid in analyzable code is dynamic memory management. Standard implementations of `malloc` do not have execution time bounds. A better solution for some dynamic memory management is using pools with a bounded size [26].

3.2 Benefits of LF for WCET Analysis

In Lingua Franca, applications are developed modularly as networks of communicating reactors, where each reactor defines reactions to individual events. The reaction bodies tend to be small pieces of code, making them more manageable by static analysis tools. Moreover, the program specifies real-time requirements by attaching deadlines to the reactions. Most importantly, the program explicitly specifies the dependencies between reactions, so the analysis tool knows every piece of code that can affect the ability to meet the deadlines. The LF syntax encourages breaking apart complex application code into a set of simple reactions, which helps generate WCET values from timing analysis tools. In addition, LF's deterministic semantics enable the generation of predictable quasi-static schedules, which help analyze timing behavior at the system level, given individual WCETs of reactions.

It is also known *how* it affects the ability to meet a deadline. A piece of code may need to be executed before the deadline expires, or it may only need to be completed before the next event arrives. Ignoring that code in certain circumstances may be reasonable in the latter case. It may, for example, be performing logging functions that utilize the difficult-to-analyze `printf` function.

For example, suppose that an arriving event triggers several reactions, that some reactions have deadlines or are dependent on reactions that have deadlines, and some do not. Then, if we assume that reactions with deadlines will be prioritized in some specified manner, we can focus the WCET analysis on the bodies of those reactions. In principle, the reactions without deadlines can be deferred indefinitely, although doing so could create performance or memory problems. Those can be guarded against by, for example, dropping or modifying log entries when problems arise. Hence, our technique's ability to include some code that is more difficult to analyze in a program makes it much more practical than techniques that require full modeling of every part of the application.

3.3 Challenges with Dynamic Scheduling

The timing behavior of some functions in the standard LF runtime is not yet analyzable using Platin [16]. These functions typically include print statements or allocate memory with `malloc` or similar memory management functions.

By default, LF programs are scheduled by a dynamic scheduler, which maintains an event queue at runtime and ensures that events are processed in timestamp order. However, the dynamic scheduler presents challenges in timing analysis. The first challenge is the use of dynamic memory allocation. For example, a common user-facing library function is `lf_set`, which sets the value of an output port of a reactor and calls `calloc` when no events allocated on the heap can be recycled. A standard solution in real-time systems is to use only statically allocated objects. A program-managed pool of preallocated objects can be used if dynamic buffer management is needed. To enable WCET analysis of standard LF programs, we propose changing the runtime to use explicit memory management.

We need to analyze individual reactions and their scheduling, which affects the overall timing behavior. Since the dynamic scheduler makes all decisions at runtime, predicting its behavior at compile time requires building an accurate model that captures its intended behavior, which is challenging.

The above challenges motivate an alternative technique for scheduling LF programs amenable to timing analysis at compile time and suitable for hard real-time systems, which we will discuss next.

3.4 Quasi-Static Scheduling of LF Applications

It is common practice for safety-critical systems to use a static schedule, usually called a cyclic executive. The pros and cons of cyclic executives have been discussed [21]. The main disadvantage of a cyclic executive is that long-running tasks often need to be split into smaller tasks to construct a feasible, static schedule. This restriction can be overcome by using multiple processor cores and scheduling long-running tasks on a dedicated processor core [29].

Recently, Lin et al. [18] developed a technique for generating quasi-static schedules from LF programs. Quasi-static schedules are encoded into bytecode programs, composed of an instruction set developed for *PretVM*, a virtual machine executing the schedules within the LF runtime. The schedules are "quasi-static" instead of "static" because parts of them can be enabled or bypassed depending on the execution context. Compared to the user-written LF reactions, which represent the *application* logic, a quasi-static schedule represents the *coordination* logic of an LF program, encoding scheduling decisions satisfying task dependencies and timing constraints. LF's quasi-static scheduling is experimental and limited to timer-driven programs. Yet, as we will show next, it offers promising analyzability. LF's quasi-static scheduler supports multiple cores. However, this initial work focuses on a single-core system. In future work, we plan to use multiple cores to execute reactors in parallel and a network-on-chip to exchange messages between reactors [14].

Unlike the default dynamic scheduler, which collects events and determines which to process next at runtime, LF under quasi-static scheduling makes all the scheduling decisions at compile-time. The user first annotates the LF program with a WCET estimate for each reaction using the `@wcet` attribute. Then, based on LF's deterministic semantics, the compiler computes the LF program's state space, identifying various execution phases and finding a hyperperiod of reaction invocations. Once the state space is determined, a set of (unpartitioned) directed acyclic graphs (DAGs) is generated. These DAGs encode dependencies among reaction invocations and between reaction invocations in real-time. A quasi-static scheduler is invoked by the LF compiler to schedule the unpartitioned DAG and generate partitions of the DAG based on the number of workers specified by the program. From that DAG partition we produce a bytecode program using the virtual instruction set.

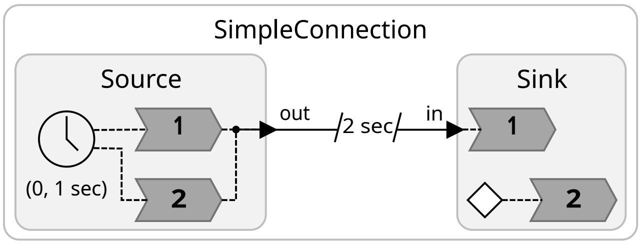

Figure 1 The graphical representation of the reactors and their connection generated by the Lingua Franca framework for the LF application shown in Listings 1 and 2.

3.5 Analyzing the Runtime System

Compiling an LF program results in a standard .elf file containing the LF runtime and the individual reactions. We need the WCET of the individual reactions to make the quasi-static schedule, which can be obtained using PLATIN. We run PLATIN on the .elf, prompting it for the WCET bound of each reaction. This is then fed to the quasi-static scheduler to try and create a schedule. For a full WCET bound of a reaction, we must also account for the time the scheduler executes. For this work, the PretVM issues specific *instructions* to schedule each reaction. These instructions take time to execute and add to the WCET of a reaction. Each instruction is executed by a dedicated function in the LF runtime. This allows us to use PLATIN again to analyze these functions to associate each instruction with its WCET bound. To calculate a reaction's WCET bound, we take the PLATIN bound and add the bounds of each instruction used to schedule the reaction, giving us a total bound for the WCET of a reaction, which also accounts for the scheduling time.

4 Evaluation

The proposed approach is demonstrated and evaluated by targeting the Patmos processor and by using the PLATIN tool for WCET analysis. Our evaluation focuses on demonstrating the feasibility and effectiveness of using LF to produce a predictable real-time application. We start by presenting a simple example application to illustrate the basic analyzability of LF reactors. Subsequently, we scale up to a medium-sized application to showcase the proposed approach's ability to handle more complex and realistic examples. For the medium-sized application, we aim to comprehensively assess the end-to-end workflow from source code to schedulability analysis, including the overhead introduced by the LF runtime.

4.1 A Simple Example Application

As an initial example for WCET analysis of a complete LF program, we use the `SimpleConnection` program, shown in Figure 1. This program has four reactions, divided into two reactors. The `Source` reactor (Listing 1) has a timer that every second triggers its two reactions. The first reaction outputs the reactor shared state variable, `s`, and then increments it. The second reaction negates the value of `s` before outputting it. The result of these reactions is that each trigger outputs the negated count that the reactor has reached. I.e., $0, -1, -2, -3$ etc. After a delay of two seconds – which is managed by the LF runtime – the `Sink` reactor's first reaction is triggered with the previous output value, which it stores in a reactor variable (Listing 2). We ignore the second `Sink` reaction as its only meant to run when the program times out.

Table 1 Individual WCET bounds in clock cycles of the reactions of Lingua Franca applications.

Application	Function	WCET Bound
`SimpleConnection`	Source reaction 1	969
	Source reaction 2	925
	Sink reaction 1	540
`ADASModel`	Brakes reaction	497
	Lidar reaction	946
	Camera reaction	946
	Dashboard reaction	497
	Processor reaction 1	1786
	Processor reaction 2	2130

Analyzing the WCET bounds of these reactions is done as previously described. In Table 1, we see the WCET bounds produced by PLATIN for each function (called "Sink reaction X" or "Source reaction X"). These numbers are given to the quasi-static scheduler, which attempts to construct a feasible schedule for them. To account for its overhead, i.e., the execution of PretVM instructions, we also analyze the functions executing those instructions and provide the numbers to the scheduler. The scheduler then uses the WCET number of its instructions and those of the reactions and attempts to construct a feasible schedule for the target, in this case, a single Patmos core. For example, the schedule would use the scheduler instructions {`EXE; ADDI`} when triggering the Sink 1 reaction. The `EXE` instruction requires 112 cycles to run (excluding the execution time of the called function), while the `ADDI` instruction requires 403 cycles. So, the cumulative execution time of Sink 1 when accounting for the scheduler is 1055 cycles.

The scheduler's output is a quasi-static schedule that, within some hyperperiod, will schedule all tasks on the available hardware. Reactions can be given deadlines that are bound by their release times. If there exists no schedule that satisfiws all deadlines, the scheduler throws an error. For example, if a reaction depended on Sink 1 with a deadline of 1000 cycles, the scheduler would trivially fail because it simply could not schedule Sink 1 early enough to meet the deadline regardless of the rest of the schedule.

4.2 Medium-sized Application

As a medium-sized application, we experimented with and modeled the "Advanced Driver Assistance System (ADAS)", a ubiquitous system in the automotive industry [19]. In this model, we have a processor that receives events from a Light Detection and Ranging (LiDAR) device and a camera and sends commands to the brakes and the dashboard. In this system, the dashboard shows a message when an object approaches the vehicle. It also triggers brakes automatically when the object is too close. In Figure 2, we model each system part as reactors connected by ports.

We again give the PLATIN-produced WCET bounds of each reaction in Table 1. Notice how the bounds for the brake and dashboard reactions are the same. This is because these reactions are only stubs, as we do not have a physical system to interact. This is also the case for the LiDAR and Camera reactions. Therefore, only the processor reactions have actual code. We use this application only to show the feasibility of implementing a real-work application, which would differ from this one only by implementing the physical end-point reactions and proper sensor fusion.

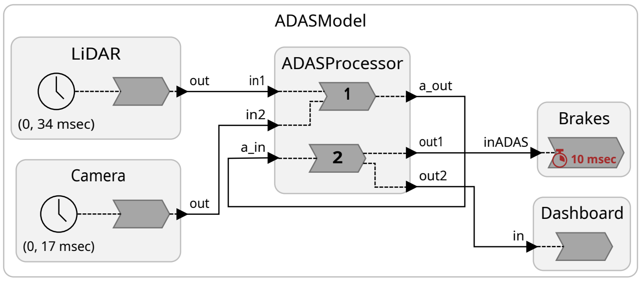

Figure 2 The graphical representation of the reactors and their connection generated by the Lingua Franca framework for the ADASModel LF application.

After we get the schedule devised by the quasi-static scheduler, we can evaluate whether it satisfies our requirements. For example, we might want to be able to detect 100 times per second whether breaking is needed using our default T-CREST test platform (Altera Cyclone IV FPGA mounted on DE2-115 evaluation board). Our FPGA runs at 80 MHz, therefore the deadline of 10 ms translates to 800 000 clock cycles. As part of the schedule construction, the overhead of each reaction is accounted for, as well as the execution time of other reactions that might run between the reaction's arrival and its beginning execution. In Figure 2, we have added the 10 ms as a deadline for releasing the brake reaction. In LF, all other tasks preceding the braking must execute within the 10 ms, cumulatively.

To check whether the schedule can meet the deadline, we look at the execution chains of the schedule. Any chain ending in the brake reaction must have a cumulative WCET bound below the 800 000 cycles. The first chain in the scheduled hyperperiod executes the following reactions: LiDAR, Camera, Processor 1, Processor 2, Brake, and Dashboard. The chain until the braking uses the following PretVM instruction counts: 3x`BEQ`, 7x`EXE`, 2x`JAL`, and 4x`ADDI`. Based on the PLATIN bounds on executing these instructions, they cumulatively add 4 086 cycles to the WCET. Coupled with the WCET of the reaction until and excluding the brake, we get a total WCET bound of 9 894 for the reactions before the braking. This is well below our limit, and so the schedule is valid. If the schedule had not been valid, the scheduler would have thrown an error, saying it could not meet the deadline. For example, we could trigger this error by reducing the deadline to 9 000 clock cycles.

In practice, the analysis must be done for all the execution chains in the scheduled hyperperiod. However, for brevity, we will omit this for the other chains in the `ADASModel` schedule.

4.3 Automated Solution

In the long term, we aim to streamline the development process for real-time systems by implementing a one-click automated solution that integrates compilation, WCET analysis, schedulability analysis, and scheduling. The automated solution will start by compiling the reaction code and the LF runtime functions (which include the functions executing scheduling instructions.) This code is analyzed using PLATIN to get the WCET bound of the reactions and LF runtime functions. Next, the LF quasi-static scheduler organizes reaction execution (classically called tasks) based on the provided WCET bounds and the deadlines and periods according to the use case's constraints. Finally, the scheduler performs schedulability analysis automatically to ensure a feasible schedule exists before creating one. Thus, it produces a schedule as an object file linked with the previously produced code to become the final application executable (ELF file). All these steps have been carried out manually for this paper. We aim to automate that process to ensure that only a correct-by-construction solution is output, meaning the resulting executable will adhere to all specified time constraints.

5 Related Work

Several projects aim to build time-predictable processors. One example is FlexPRET [36], the latest version of the so-called precision timed machines [7]. FlexPRET also aims to be a platform that supports LF applications. FlexPRET includes timing instructions for the precise timing of reactions. Furthermore, FlexPRET implements fine-grain multithreading. FlexPRET is not yet supported by any WCET analysis tool. However, as PLATIN now also supports the RISC-V architecture [25], we will be able to adapt PLATIN for FlexPRET. Like the multicore version of Patmos, the InterPRET [14] projects aim for a multicore version of FlexPRET supporting parallel execution of LF actors on multiple cores.

ForSyDe (Formal System Design) [30, 31] is a methodology enabling high-level abstraction modeling and design of heterogeneous systems-on-chip and cyber-physical systems. The idea is to integrate formal methods from the specification phase and use formal refinement techniques to bridge the gap between specification and implementation. Thus, creating a correct-by-design system. The most interesting aspect related to our solution is the ability to employ formally analyzable models of predictable platforms and applications to provide service guarantees, which are essential in time-predictable applications.

MIRSA C [3] is a set of software development guidelines for the C programming language to ensure that C code is safe, reliable, and maintainable. Even if these guidelines do not directly address time-predictable systems, enforcing a strict coding standard helps avoid undefined behaviors that can lead to unpredictable execution times. One very concrete guideline is the prohibition of the use of dynamic memory allocation functions such as `malloc()`, `calloc()`, and `free()`. These are restricted to avoid memory leaks, fragmentation, and unpredictable behavior. The latter is particularly relevant for real-time systems since the time taken to allocate or deallocate memory can vary significantly and can be difficult to predict.

The review presented in [35] discusses the current challenges in WCET analysis and surveys several WCET tools, highlighting their methods, functionalities, and limitations. Here, two main categories of WCET tools are identified: static analysis and measurement-based or hybrid tools. Static analysis tools determine WCET by analyzing the code without executing it. These tools construct a detailed model of the program and the processor to estimate execution time bounds. They mainly focus on control-flow and data-flow analysis to provide guaranteed upper bounds on execution times. Measurement-based or hybrid tools (using measurements and static analysis) estimate the WCET by executing the program or its parts on actual hardware or simulators. They measure execution times of code segments and use these measurements to infer timing bounds. This approach often results in more accurate estimates for complex systems but may lack formal guarantees. Commercial tools examples include aiT [1, 9] from AbsInt, Bound-T from Tidorum [2, 12], and RapiTime from Rapita Systems [4]. Examples from academia include Heptane [8], Chronos [17], and SWEET [20].

6 Conclusion

This paper presented the integration of the time-predictable processor Patmos with the Lingua Franca (LF) coordination language and the Platin WCET analysis tool. More specifically, we used the WCET analysis tool PLATIN to analyze individual reactions of LF reactors and the runtime of the quasi-static schedule. Using a simple and a medium-sized application, our evaluation confirmed that carefully written LF programs can be analyzed for WCET, creating a quasi-static schedule that fulfills all timing requirements for safety-critical applications.

References

1 aiT webpage. Online at `https://www.absint.com/ait/`. Accessed: 07 June 2024.

2 Bound-T webpage. Online at `http://www.bound-t.com/`. Accessed: 07 June 2024.

3 MISRA webpage. Online at `https://misra.org.uk/`. Accessed: 07 June 2024.

4 RapiTime webpage. Online at `https://www.rapitasystems.com/products/rapitime`. Accessed: 07 June 2024.

5 Lingua franca website:. `http://www.lf-lang.org/`, 2024.

6 Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE. `doi:10.1109/ISORC.2014.47`.

7 Stephen Edwards and Edward Lee. The case for the precision timed (PRET) machine. In *2007 44th ACM/IEEE Design Automation Conference*, pages 264–265, July 2007. URL: `https://ieeexplore.ieee.org/abstract/document/4261184`.

8 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *Open Access Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/OASIcs.WCET.2017.8`.

9 Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH, 2013. [Online, last accessed November 2013].

10 Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

11 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

12 Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Bound-T time and stack analyzer, reference manual. Technical report, Tidorum Ltd, 2013. available at `http://www.bound-t.com/manuals/ref-manual.pdf`.

13 Benedikt Huber, Stefan Hepp, and Martin Schoeberl. Scope-based method cache analysis. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 73–82, Madrid, Spain, July 2014. `doi:10.4230/OASIcs.WCET.2014.73`.

14 Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. Interpret: A time-predictable multicore processor. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, pages 331–336, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3576914.3587497`.

15 Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 55–64, New York, NY, USA, 2013. ACM. `doi:10.1145/2516821.2516828`.

16 Ehsan Khodadad, Luca Pezzarossa, and Martin Schoeberl. Towards lingua franca on the patmos processor. In *2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, 2024.

17 Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007. Special issue on Experimental Software and Toolkits. `doi:10.1016/j.scico.2007.01.014`.

18 Shaokai Lin, Erling Jellum, Mirco Theile, Tassilo Tanneberger, Binqi Sun, Chadlia Jerad, Ruomu Xu, Guangyu Feng, Christian Menard, Marten Lohstroh, Jeronimo Castrillon, Sanjit Seshia, and Edward Lee. Pretvm: Predictable, Efficient Virtual Machine for Real-Time Concurrency, 2024. `arXiv:2406.06253`.

19 Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. Towards building verifiable cps using lingua franca. *Acm Transactions on Embedded Computing Systems*, 22(5 s):155, 2023. `doi:10.1145/3609134`.

20 Björn Lisper. Sweet – a tool for wcet flow analysis. In Bernhard Steffen, editor, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer-Verlag, October 2014. URL: `http://www.es.mdu.se/publications/3693-`.

21 C. Douglas Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

22 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *Acm Transactions on Embedded Computing Systems*, 20(4):36, 2021. `doi:10.1145/3448128`.

23 Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. A language for deterministic coordination across multiple timelines. *Forum on Specification and Design Languages*, 2020-:9232939, 2020. `doi:10.1109/FDL50818.2020.9232939`.

24 Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. Actors revisited for time-critical systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 152:1–152:4, New York, NY, USA, June 2019. ACM. `doi:10.1145/3316781.3323469`.

25 Emad Jacob Maroun, Eva Dengler, Stefan Dietrich, Chistian Hepp, Benedikt Herzog, Henriette Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann. The platin multi-target worst-case analysis tool. In *22th International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, 2024.

26 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 79–88. IEEE, 2004.

27 George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998. `doi:10.1016/S0065-2458(08)60208-9`.

28 Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

29 Anders P. Ravn and Martin Schoeberl. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience*, 24:772–788, 2012. `doi:10.1002/cpe.1754`.

30 Ingo Sander. *System Modeling and Design Refinement in ForSyDe.* PhD thesis, KTH, Microelectronics and Information Technology, IMIT, 2003. NR 20140805.

31 Ingo Sander and Axel Jantsch. Modelling adaptive systems in forsyde. *Electronic Notes in Theoretical Computer Science*, 200(2):39–54, 2008. Proceedings of the First Workshop on Verification of Adaptive Systems (VerAS 2007). `doi:10.1016/j.entcs.2008.02.011`.

32 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. `doi:10.1016/j.sysarc.2015.04.002`.

33 Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, April 2018. `doi:10.1007/s11241-018-9300-4`.

34 Robert Tolksdorf. Models of coordination. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1972:78–92, 2000. `doi:10.1007/3-540-44539-0_6`.

35 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008. `doi:10.1145/1347375.1347389`.

36 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, Berlin, Germany, April 2014.