Navigating Time and Energy Trade-offs in Reactive Heterogeneous Systems

Shaokai Lin, *UC Berkeley*
Tassilo Tanneberger, *TU Dresden*
Jiahong Bi, *TU Dresden*
Guangyu Feng, *UC Berkeley*
Yimo Xu, *UC Berkeley*
Julian Robledo, *TU Dresden*
Robert Khasanov, *TU Dresden*
Jeronimo Castrillon, *TU Dresden*

***Abstract*—Reactive software poses challenging requirements: deterministic execution with stringent timing constraints under a tight energy budget. Meeting these requirements is particularly hard when executing on the increasingly heterogeneous platforms of today. In this paper, we integrate MOCASIN, a design space exploration tool, into LINGUA FRANCA, a programming framework for building deterministic and timed reactive software. We show that this integration enables choosing a desired timing and energy performance at design time. We demonstrate our approach in satellite attitude control, consisting of periodic real-time tasks and sporadic non-real-time tasks. The latter sporadic tasks are coordinated using quasi-static schedules, computed by MOCASIN, leading to less energy consumption compared to the Linux scheduler under CPU frequency scaling governors such as `powersave`, `schedutil`, and `ondemand`.**

***Index Terms*—Design Space Exploration, Compiler, Quasi-Static Scheduling, Concurrency, Energy Consumption**

I. INTRODUCTION

Reactive cyber-physical systems (CPSs) have stringent real-time constraints and other practical considerations, such as energy, memory, heat, size, cost, etc., making them particularly challenging to design. Barring other considerations, it is hard to ensure that real-time constraints alone can be satisfied. Thus, a designer typically resorts to a *bottom-up* design flow, as noted by Henzinger and Kirsch [1], in which an implementation is produced first based on intuition, then it undergoes testing. If the timing behavior violates constraints, the designer revises the implementation and tests it again. The process repeats until no violations are observed. This bottom-up process can be time-consuming and error-prone, since a code change fixing one constraint might violate others. In contrast, a *top-down* approach checks if such constraints can be satisfied at design time, only if so, a correct-by-construction implementation is then generated. This reduces the time to market and ensure correctness of the design. Top-down approaches are widely adopted in the EDA domain, where a circuit's timing, area, and power can be predicted in the design tool, making it feasible to coordinate billions of transistors reliably. In the CPS domain, top-down approaches exist for models such as SDF [2]. Yet, for emerging programming models like LINGUA FRANCA (LF) [3], which facilitates the design and implementation of deterministic, real-time, and concurrent systems, providing top-down methods addressing timing and energy remains an open problem. This work aims to step towards solving this problem.

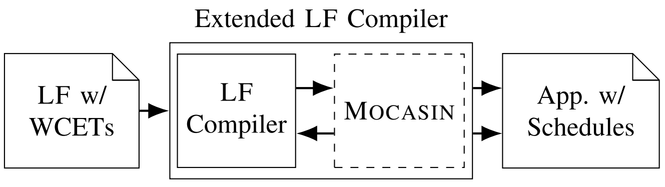

Fig. 1: Our proposed tool flow. Dashed boxes denote external tools we build interfaces for. App. stands for application logic.

Energy consumption is an important dimension to consider, as many mission-critical CPSs are powered by batteries. For resource-constrained systems such as satellites, drones, and sensor networks, new techniques for energy management becomes ever more important due to recent trends of adopting heterogeneous multicores. Even for systems that are not considered resource-constrained in the traditional view, such as self-driving cars, energy management is increasingly a concern due to the deployment of energy-hungry hardware, *e.g.,* GPUs.

We make the following contributions in this work:

1) We extend the quasi-static scheduling approach of the LINGUA FRANCA compiler [4] by exploring performance-energy tradeoffs using MOCASIN. Our extension takes in an LF program with worst-case execution time (WCET) annotations, and outputs application logic and quasi-static schedules mapping LF reactions to a heterogeneous hardware platform, as shown in Fig. 1.
2) We present a case study on implementing a three-axis reaction wheel controller, used for satellite attitude control. We show that the sporadic non-real-time tasks coordinated by quasi-static schedules, computed by MOCASIN, yield less total energy consumption compared to the Linux scheduler under CPU frequency scaling governors such as `powersave`, `schedutil`, and `ondemand`.

II. BACKGROUND

The rise of the multi- and many-core era and the trend for integrating an increasing amount of heterogeneous computing resources, interconnects, and memories in a single multiprocessor system on a chip (MPSoC) has motivated the development of a multitude of design space exploration (DSE) tools.

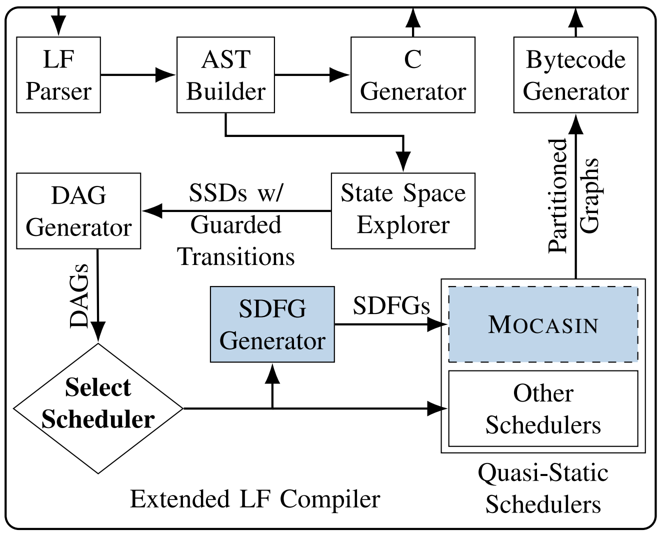

Fig. 2: Overview of our extension based on [4]. Colored boxes are our work.

DSE tools aim to increase software productivity by abstracting hardware complexity and automating the process of mapping software components to hardware components [5]. MOCASIN is a recently proposed DSE framework [6], which provides a comprehensive platform model that captures heterogeneous hardware on a relatively high abstraction level, supports applications in various input formats and using SDF or Kahn process network (KPN), implements various heuristics and meta-heuristics known from the literature, and provide a high-level simulator that allows to estimate the make span and the energy consumption of a given application on the target hardware when using a specific mapping.

LINGUA FRANCA, on the other hand, is a programming framework, in which the user can define and compose reactive components called *reactors*, *i.e.*, stateful containers with event-triggered message handlers called *reactions* written in target languages including C, C++, Rust, Python, and TypeScript. In this work, we use the C target. While by default LF uses a dynamic scheduler [3], this work builds on a recently introduced quasi-static scheduling technique for LF [4]. Instead of using an event queue to track new tasks at runtime, [4] computes a program state space at compile time and generates a schedule that quasi-statically encodes when tasks are launched. The schedules are written using an instruction set for a virtual machine called PRETVM.

III. DSE FOR LF BASED ON PRETVM

In this section, we present our end-to-end approach for compiling an LF program into PRETVM bytecode while enhancing the execution time and energy under the guidance of a MOCASIN-based scheduler.

Key Assumptions: Our approach relies on several key assumptions: (a) The chosen hardware is supported by MOCASIN. (b) The user has the ability to obtain *estimates* on the WCETs of C functions. Most of the time, the estimates could be obtained through direct measurements repeated for a sufficient number of times. For certain platforms, it is possible to use commercial-off-the-shelf timing analysis tools, such as AbsInt [7]. We emphasize here that we treat these input execution times as estimates, instead of the ground truth. This is sufficient for proving the property that "the program can provably meet the hard real-time constraints, *given that the estimates hold at runtime*." (c) The LF program under analysis can be executed under LF's fast mode, meaning that reactions do not wait for physical time points to be released. (d) The LF program only contains logically periodic tasks, *i.e.*, driven by LF timers (same assumption as in [4]).

A. Preparing Inputs

The user first prepares an LF program with worst-case execution time information using the `@wcet` attribute. As an example, Fig. 3 shows a reaction from our case study with a `@wcet` annotation. Here, the `@wcet` annotation contains two values since we target the Odroid-XU4, a platform using Arm big.LITTLE architecture with two different types of CPU cores. The first number represents the WCET of the reaction on the LITTLE core (Cortex-A7) while the second number represents the WCET on the big core (Cortex-A15). We plan to support alternative input methods in the future, such as loading WCETs from a configuration file.

```
@wcet("7797 nsec, 5294 nsec")
reaction(done_c1,done_c2,done_c3) -> start_out,done_out {=
    if (self->count >= self->num_files) {
        lf_set_present(done_out);
    } else {
        lf_set_present(start_out);
    }
=}
```

Fig. 3: Reaction with a `@wcet` annotation.

B. Generate and Register Mappings

Fig. 2 shows the internals of the LF compiler with our extension highlighted in blue. The compiler parses and transforms the input program into an abstract syntax tree (AST). A model of the LF program is created from the AST and fed into a State Space Explorer, which simulates and represents the program state space as a set of state space diagram (SSDs) [8]. The SSDs are further transformed into a set of Directed Acyclic Graphs (DAGs), each corresponding to a phase of LF execution (e.g., initialization, periodic, and shutdown phase). The role of the quasi-static schedulers is to assign tasks to workers by partitioning the DAGs [4].

In this work, we implement an SDFG generator, which generates synchronous dataflow graphs (SDFGs) out of input DAGs, by drawing an additional edge from the tail node of the DAG to the head node. The generated SDFGs are then sent to a quasi-static scheduler we implemented based on MOCASIN, within which MOCASIN invokes the `pareto_front` command on the SDFG for the periodic phase. MOCASIN runs design space exploration (DSE) and outputs a mapping file enumerating a list of candidate mappings between tasks and CPU cores, which MOCASIN projects to yield the best timing and energy performance. The user can choose from the candidate mappings an ideal mapping, e.g., one projected to consume the least energy, and register the mapping into the original LF program using the `mocasin-mapping` target property field.

C. Generate and Execute Schedules

Table I shows a subset of mappings that MOCASIN generated after design space exploration. In the generated file, each mapping maps a task to a particular core, with execution time and total energy estimated by MOCASIN.

Tasks	Mapping 0	Mapping 1	Mapping 2
c1.reaction_1	Core 7 (A15)	Core 6 (A15)	Core 2 (A7)
c2.reaction_1	Core 6 (A15)	Core 6 (A15)	Core 2 (A7)
c3.reaction_1	Core 4 (A15)	Core 5 (A15)	Core 2 (A7)
d.reaction_2	Core 4 (A15)	Core 4 (A15)	Core 2 (A7)
d.reaction_3	Core 7 (A15)	Core 4 (A15)	Core 2 (A7)
d.reaction_4	Core 5 (A15)	Core 4 (A15)	Core 2 (A7)
Exec. time (s)	41.914	83.828	312.753
Total energy (J)	257.458	348.985	783.102

TABLE I: A subset of mappings for the periodic phase.

Once the selected mapping is registered back into the LF program, the scheduler pools all tasks mapped to the same CPU core into one partition for a worker. Then for each worker, the LF compiler combines the worker's workload across different execution phases, forming a single quasi-static schedule for the worker. From each quasi-static schedule, a sequence of virtual instructions is generated for the worker [4].

As a result of compilation, LF generates programs and useful utilities that come from the original inputs. Then the user can pin different workers to the corresponding cores based on the user-selected mapping. For example, if worker thread m's schedule contains tasks meant to be executed on Core n, as suggested by MOCASIN, then the user should pin worker thread m to Core n, which can be achieved on Linux by setting CPU Affinity and using the `cpuset` infrastructure.

IV. CASE STUDY

A. Application

For our case study, we develop a realistic satellite attitude control application in LF, which has two parts: the first is a real-time reaction wheel controller based on [9]; the second is a sporadic file compression service, which compresses data files before they are sent back to earth. Each data file is further assumed to have a maximum size. The source code is publicly available on GitHub.[1] In this case study, we only focus on the file compression service.

The CompressFiles service consists of three `Compressor` reactors that implement the primary compression logic. In addition, there is a `Director` reactor that dispatches incoming jobs to one of the three `Compressor` reactors. Within the `Director` reactor, reactions 2 and 3 are of particular interest. Reaction 2 collects all `done` signals from the three

[1]https://github.com/icyphy/satellite-attitude-control

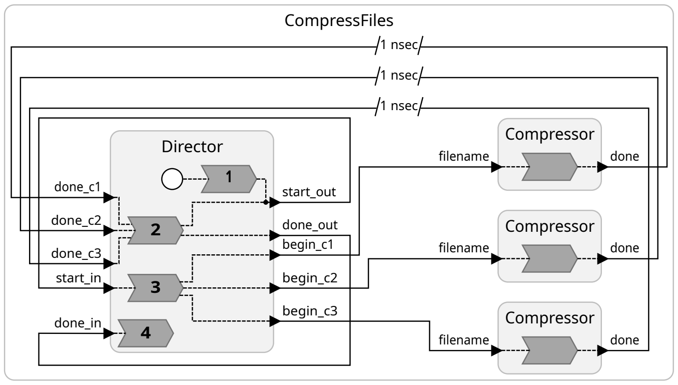

Fig. 4: LF diagram of the file compression service.

`Compressor` reactors and determines whether processing can be stopped, in which case the `done_out` signal is set, otherwise the `start_out` signal is set. Reaction 3 reacts to the start signal described above and pushes a new set of files to be compressed to the `Compressor` reactors.

B. Experimental Setup

We run the experiments on the Hardkernel Odroid-XU4 platform featuring an Exynos 5422 big.LITTLE chip with four Cortex-A15 ("big") and four Cortex-A7 ("LITTLE") cores. The frequency ranges are set to $200\,\mathrm{MHz}$ to $2.0\,\mathrm{GHz}$ for "big" cores, and $200\,\mathrm{MHz}$ to $1.4\,\mathrm{GHz}$ for "LITTLE" cores. We measured the energy consumption of the Odroid-XU4 board using ZES Zimmer LMG450 Power Analyzer connected to DC input with an external readout rate of $20\,\mathrm{Sa/s}$. The Linux kernel version we use is 5.4.228-412.

The main method of controling the power consumption of a CPU is frequency scaling, apart from shutting down individual cores. Linux implements a concept called *governors*, which implements different strategies for scaling the CPU frequency. The relevant governors we evaluate are:

- **`Performance`** Sets the frequency to the highest possible.
- **`Powersave`** Sets the frequency to the lowest possible.
- **`Ondemand`** Scales the frequency based on the current load.
- **`Schedutil`** Scales the frequency based on the CPU utilization info from the scheduler, and uses Energy Aware Scheduling (EAS) to map tasks to cores to minimize energy consumption on heterogeneous platforms.

C. Measurement Results

We perform five sets of experiments using the LF program `CompressFiles`. The first experiment uses the quasi-static scheduler with a task-to-core mapping computed by MOCASIN, and the rest of the experiments use LF with the default dynamic scheduler coupled with the Linux governors introduced in Section IV-B, representing the common strategies a regular Linux user would use to minimize energy. For each configuration, the service compresses 15 CSV files, and measurements were taken based on the average of one hundred repeated program executions. In the first experiment, MOCASIN explores the design space assuming constant CPU

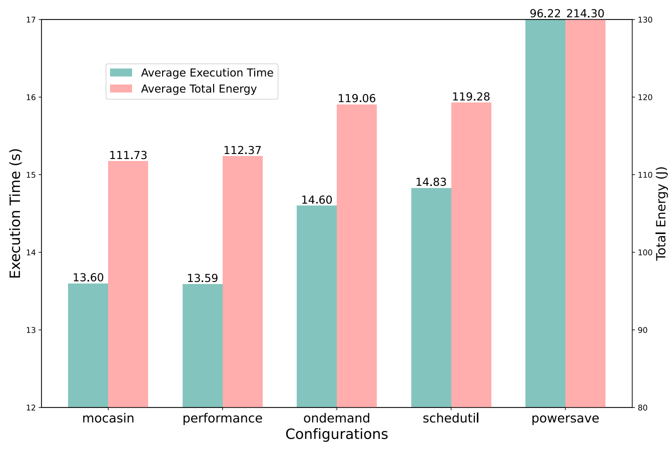

Fig. 5: Execution time and total energy measured.

frequencies; to satisfy this assumption, the **performance** governor is used. In addition, we pin reactions to each type of CPU core under a fixed frequency and use LF's tracing utility to collect the reactions' core-specific WCET estimates.

Table I shows a subset of mappings generated by MOCASIN in the first experiment. We choose to use mapping 0 as it is projected to consume a relatively small amount of energy among all available mappings without compromising execution time.

Fig. 5 shows the measured execution time and total energy from all five experiments. While the **powersave** governor does exhibit the least power (in terms of wattage), since it sets each CPU core to the minimum frequency (200 MHz), it significantly prolongs the execution time (96.22 s) and, in fact, yields the *largest* total energy consumption (214.3 J) among the experiments. This result highlights the fact that saving power is not the same as saving total energy.

The **ondemand** and **schedutil** governor (which uses EAS) share similar execution times and total energy. They perform better than **powersave** since they dynamically scale the CPU frequencies based on the workload. The shorter execution times result in less total energy consumed. MOCASIN and **performance** turn out to have the best performance in the end, since big cores are prioritized in both strategies, resulting in two of the smallest execution times as well as total energy consumed. This could seem counter-intuitive to regular users who choose **powersave** or **schedutil** to conserve energy.

D. Discussion

While using big cores seems to minimize execution time and total energy, the choice of cores depends on additional factors. In the space domain, these factors include mission specifications, solar power charging rate, and battery size. Here, we focus on the *total* (static + dynamic) energy consumption, instead of the *dynamic* energy alone, on the assumption that the hardware on the satellite enters a low-power or non-consuming state after completing a computational task, rendering static energy consumption negligible in this context. If the mission restricts low-power mode, the **performance** governor (using big cores) would not be ideal due to high static energy consumption. A low solar power charging rate and small battery may also make big cores impractical. In such cases, trading execution time for power efficiency with LITTLE cores might be better. A top-down design technique, such as our proposed LF extension, could better handle such design complexities as design space exploration could factor in these constraints upfront. On the other hand, bottom-up approaches could be error-prone, as we have seen with the **powersave** case, and inefficient, as it often requires trials and errors.

V. CONCLUSION

In this work, we extend the quasi-static scheduling technique for LF with support for interfacing with MOCASIN, which enables DSE for a subset of LF programs on heterogeneous platforms and generating schedules that encode user-specified trade-offs between execution time and total energy consumption. The case study shows that our approach effectively conserves total energy consumption compared to alternatives using Linux frequency scaling governors.

ACKNOWLEDGMENT

This work was supported by the German BMBF programme "Souverän. Digital. Vernetzt." (6G-life, grant 16KISK001K) and E4C (grant 16ME0426K), the EU Horizon Europe Programme under grant agreement No 101135183 (MYRTUS), the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems).

The authors also thank Linh Thi Xuan Phan, Martin Schoeberl, and anonymous reviewers for their feedback and suggestions, which help improve this work.

REFERENCES

[1] T. A. Henzinger and C. M. Kirsch, "The embedded machine: Predictable, portable realtime code," in *International Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002, Conference Proceedings, pp. 315–326.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, may 2021.

[4] S. Lin, E. Jellum, M. Theile, T. Tanneberger, B. Sun, C. Jerad, R. Xu, G. Feng, C. Menard, M. Lohstroh, J. Castrillon, S. Seshia, and E. Lee, "PretVM: Predictable, Efficient Virtual Machine for Real-Time Concurrency," 2024.

[5] J. Castrillon Mazo and R. Leupers, *Programming Heterogeneous MP-SoCs: Tool Flows to Close the Software Productivity Gap*. Springer, 2014.

[6] C. Menard, A. Goens, G. Hempel, R. Khasanov, J. Robledo, F. Teweleitt, and J. Castrillon, "Mocasin—rapid prototyping of rapid prototyping tools: A framework for exploring new approaches in mapping software to heterogeneous multi-cores," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, ser. DroneSE and RAPIDO '21, 2021.

[7] "The industry standard for static timing analysis." [Online]. Available: https://www.absint.com/ait/index.htm

[8] S. Lin, Y. A. Manerkar, M. Lohstroh, E. Polgreen, S.-J. Yu, C. Jerad, E. A. Lee, and S. A. Seshia, "Towards building verifiable cps using lingua franca," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–24, 2023.

[9] J. Cardoso, C. Chanel, P. Chauvin, A. Hostallier, J. Lamaison, A. Mascarenas-Gonzales, E. Metral, and L. Alloza, "Lab on the real-time control of reaction wheel," 2022, unpublished lab exercise for the course 1MAE803: Real-time Control of Aerospace Systems, M.Sc in Aerospace Engineering, ISAE-SUPAERO.