Worst-Case Reaction Time Optimization on Deterministic Multi-Core Architectures with Synchronous Languages

Nicolas Hili, Alain Girault, Eric Jenn

Worst-Case Reaction Time Optimization on Deterministic Multi-Core Architectures with Synchronous Languages

Nicolas Hili*, Alain Girault†, and Éric Jenn*‡

* IRT Saint-Exupéry, 3 Rue Tarfaya, CS 34436, 31400 Toulouse, France, first.last@irt-saintexupery.com
† Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France, alain.girault@inria.fr
‡ Thales AVS, 105 Avenue du Général Eisenhower F31036 Toulouse, France, eric.jenn@fr.thalesgroup.com

***Abstract*—In this paper, we propose a new approach for the predictability and optimality of the inter-core communication and execution of tasks allocated on different cores of multi-core architectures. Our approach is based on the execution of synchronous programs written in the ForeC programming language on deterministic architectures called *PREcision Timed*. The originality of the work resides in the time-triggered model of computation and communication that allows for a very precise control over the thread execution. Synchronization is done via configurable Time Division Multiple Access (TDMA) arbitrations where the optimal size and offset of the time slots are computed to reduce the inter-core synchronization costs. We implemented a robotic application and simulated it using MORSE, a robotic simulation environment. Results show that the model we propose guarantees time-predictable inter-core communication, the absence of concurrent accesses (without relying on hardware mechanisms), and allows for optimized execution throughput.**

***Index Terms*—Multi-core, Architecture, PRET, Synchronous Languages, Inter-core synchronizations**

I. Introduction

Over the past decade, switching from single-core to multi-core architectures has been the natural choice in many domains in order to improve the performance of software application while reducing the Size, Weight, and Power (SWaP) of the computing platforms [1]. However, performance improvement usually comes at the cost of temporal non-determinism [2] for it relies on very complex and hardly predictable combinations of micro-architecture mechanisms (e.g., multi-level caches, branch predictors, ...). Although trading better average performance for a loss of predictability is sometimes acceptable, it is not for safety-critical applications where the time at which a value is produced is often as important as the value itself.

The issue addressed in this paper can be summarized in one question: *how to guarantee the determinism of distributed real-time applications deployed on multi-core architectures while not sacrificing the global performances of the system?* To address this issue, we propose to combine the highly-predictable *PREcision Timed (PRET)* micro-architecture with a synchronous model of communication and execution, and a Time Division Multiple Access (TDMA) arbitration mechanism for the bus connecting the different cores.

The different contributions made in this paper are:

- An approach to programming synchronous applications on multi-core architectures. It relies on a time-triggered model of communication based on *rendezvous* to guarantee task synchronization at very precise points in time. It combines analyses of Worst-Case Execution Times (WCETs) and Integer Linear Programming (ILP) to optimize the global throughput of the application given a specific platform topology.
- The hardware extension of the single-core FlexPRET [3], [4] to a multi-core version called MultiPRET, along with its simulation environment. We propose three deterministic inter-core communication mechanisms based on TDMA arbitration and we show how the time-triggered model we propose can be applied.
- The validation of the approach through a robotic application.

The synchronous language we have chosen is ForeC [5], [6], a C-based synchronous programming language developed collaboratively by INRIA and the University of Auckland. Compared to other synchronous languages, ForeC shows two singular properties. Firstly, it has been explicitly designed to support the development of applications deployed on multi-core architectures. At design time, its syntax and semantics (threads, shared variables, etc.) allows for expressing parallelism at code-level. At execution time, a ForeC program is executed by a set of truly parallel *threads* potentially executed on different cores according to a static *thread-to-core mapping*. Secondly, its syntax reuses and extends C, which makes it easier to learn for developers who are familiar with C.

The paper is structured as follows: Section II presents different deterministic architectures and models of programming for real-time critical systems; Section III presents the execution model of ForeC and introduces our running example; Section IV details our approach, formalizes the time-triggered model of communication, and applies it to different architectures; Section V describes the extension made to FlexPRET and details the evaluation we conducted; Finally, Section VI concludes the paper.

II. Background

A. Deterministic Hardware Architectures

To achieve a highly-predictable micro-architecture, one way consists in giving up some of the mechanisms found in most

Component Off the Shelf (COTS) processors. For instance, the *PREcision Timed* family of micro-architectures introduced by Lee et al. [7] are exempt from bus congestion management, cache memories, and control hazard mechanisms. Despite the absence of those mechanisms, an acceptable level of performances is maintained thanks to a fined-grained (hardware-level) multi-threaded RISC pipeline and fast one-cycle access ScratchPad Memories (SPMs).

Several flavours of PRET architectures have been designed, implementing different Instruction Set Architectures (ISAs) [3], [5], [8], [9]. ARPRET [5] and FlexPRET [3], [4] are the latest of this series. The latter targets mixed-critically applications where Hard Real-Time Threads (HRTTs) and Soft Real-Time Threads (SRTTs) can be interleaved, while keeping a strict temporal and spatial isolation between them. FlexPRET uses the open RV32I ISA [10]. Programs are written in C and compiled using GCC RISC-V[1]. FlexPRET is, however, a single-core processor. To the difference of FlexPRET, ARPRET [5] is a reactive multi-core processor implemented as a soft-core on a Xilinx Microblaze [11]. It can execute applications written in synchronous languages, including PRET-C [12] and ForeC [5]. More details are given in Sections II-C and III.

B. Deterministic Communication Architectures

One of the main challenge of multi-core architectures is to support resilient, high-performance, and time-predictable communication architectures to deal with shared resources accessed by tasks running in parallel. TDMA arbitrations are often favoured for multi-core architectures due to their simplicity and their highly deterministic timing behaviour [13], [14]. However, they are often resource-inefficient since several time slots assigned to a core can be wasted when the core does not perform memory accesses. Various static and dynamic scheduling policies have been proposed in order to optimize the use of the time slots [13], [15]–[17]. Our work is inspired from them. In our work, we focus on pure timed-triggered models of communication [18] supported by PRET architectures, meaning that optimal time slots can be configured at compile-time [19].

TDMA schemes have been successfully combined with PRET architectures. In ARPRET, thread communications use global variables located in a shared memory accessible via a TDMA bus [5]. The TDMA bus has fixed-length time slots accessed by the different cores in a round-robin fashion. However, the TDMA bus is not optimized for a specific application, therefore a thread deployed on a specific core may require access to the TDMA bus outside of its allocated slot, causing the bus access to be delayed until the next allocated slot. Delaying is done using a `receive` routine that blocks the execution of all threads on the core, preventing the core from executing other (potentially less critical) tasks.

In a purely timed-triggered fashion, the initial implementation of a PRET architecture at UC, Berkeley [8] introduced the concept of a *memory wheel*. While not intended for inter-core communication, the memory wheel acts like a TDMA bus in order to grant different threads access to a shared address space of an off-chip memory. When the memory wheel is aligned with a memory access, accessing the off-chip memory is performed in 13 cycles. Otherwise, it waits up to another 77 cycles[2]. A `DEAD` x instruction is used to pause a thread for x cycles, in order to align its execution with the memory wheel. Placing the `DEAD` instructions and computing the optimal values of the x to get the perfect alignment ensures that all memory accesses never block, but is difficult, all the more when the number of threads becomes significant. In our work, we propose to combine both approaches [8] and [5].

C. Deterministic Software Architectures

Enforcing determinism at hardware-level is a necessary condition to achieve determinism at application-level, but it is not sufficient: the software must also behave deterministically. Towards that goal, various deterministic models of execution have been proposed for real-time distributed programming [20]–[26]. They can be categorized into three categories [23]: Zero Execution Time (ZET), Logical Execution Time (LET), and Bounded Execution Time (BET).

Synchronous languages [26] are parts of the ZET category. They allow programmers to abstract away temporal constraints, hence facilitating the design and the (formal) verification of the produced applications. Synchronous languages rely on the “synchrony hypothesis”, i.e., between the start and the end of a processing (called a *tick*), no modification of the state of the system can be observed, hence the tick is conceptually done instantaneously, in *zero-time*. Naturally, this *instantaneity* is conceptual and any computation will actually take some physical time to complete, but as long as the computation completes before the occurrence of the next external event, the conceptual model is an appropriate abstraction of reality (i.e., the synchronous hypothesis is satisfied) and the programmer does not have to care about the physical time at which any particular operations takes place. Finding the longest computation results in finding the Worst-Case Reaction Time (WCRT) of the system.

Despite the benefits of synchronous languages, C, together with Ada [24], remains the predominant programming language used for real-time system development [20]. Support of concurrent execution models and timing constructs are provided by external libraries (e.g., Posix). Other C-based programming languages have been proposed, including Real-Time Concurrent C [25] and more recently Timed C [27]. C-based programming languages are often favoured thanks to their portability to various platforms, including UNIX platforms, RTOSs, and bare-metal. Besides, they are familiar to most programmers and are strongly supported by compilers and toolchains. The problem is their lack of abstract constructs to manipulate *physical time*, which is done thanks to external functions, which makes the development of real-time systems complex and fragile.

In our work, we combine the benefits of both worlds by choosing ForeC [5], a C-like synchronous language dedicated for multi-core programming allowing us to reuse the C syntax.

[1] A variant of The GNU Compiler for RISC-V architectures.

[2] Worst case reached when a thread just missed its allocated time slot. For 6 threads and 13 cycles for performing memory access: $13 * 6 - 1 = 77$.

D. Motivation

To summarize, we propose to combine the approaches of PRET [8] and ForeC [5]. Using synchronous languages, formal analyses can be easily performed to precisely determine the WCRT of the application. However, unlike [5], inter-core communication is achieved and optimized via TDMA bus arbitration to access to the shared memory, where time slots are configured in order to be aligned with the memory accesses of the threads. This enables the size and offset of the TDMA slots to be computed from the knowledge of the execution time of the threads interleaved on the different cores, so as to optimize the global throughput of the application. In our approach, memory accesses always succeed since memory operations always occur within the TDMA allocated time slots. To achieve this, we follow the approach of [8] to position *delays* to pause the execution of a thread until it can access the shared memory. Delays are expressed in absolute time, so they are computation path-independent. Computing these delays is automatically done offline through ILP as described in Section IV. Using a synchronous model of computation facilitates the positioning of these delays as shared memory accesses only occur at the start and at the end of a tick.

Our contribution is based on FlexPRET. Compared to the Microblaze implementation of [5], FlexPRET supports two modes of execution: *single-threaded* and *multi-threaded*. In the multi-threaded mode, threads are interleaved, therefore, when a thread pauses, it can free up some CPU cycles for other threads, potentially less critical, to execute. This allows mixed-criticality applications combining SRTTs and HRTTs.

III. ForeC in a Nutshell

A ForeC program is composed of a set of threads that are created during the execution of `par` statements. The thread that executes the `par` statement is the *parent* of the threads created by the statement. So, the set of threads is a tree rooted on the thread that has executed the application's entry point. Threads communicate with each other via variables. Variables can be of four kinds. Input (resp. output) variables are variables sampled from (resp. emitted to) the environment. They are declared at the top level of the program and are accessible by all threads. Variables declared with the `shared` keyword are shared between a parent thread and its children. Normal variables can also be used.

Conceptually, a ForeC program executes at the cadence of the *global tick*. At the beginning of the global tick, all input variables are sampled from the environment. Then, every thread starts its *local tick*, which consists in: 1) creating copies of shared variables they can access, 2) performing the thread's operations; and 3) propagating the local copies of the variables to their parent thread. Operations are performed on local copies of shared variables to avoid concurrent accesses and race conditions. If multiple threads update the same shared variables, a *combination* function must be specified to combine all values computed by the threads. A combination function must be associative and commutative, so the combination is order-independent. During its local tick, the `par` statement is used to fork-join threads. A local tick ends when a `pause` statement is reached. A global tick ends when all local ticks end. At the end of the global tick, output variables are emitted to the environment.

The way ForeC threads are actually executed in parallel depends on the *target platform* on which the program is executed. ForeC currently supports three architectures: two bare metal platforms (Xilinx multi-core Microblaze [11] and PTARM single-core multi-threaded PRET [9]), and x86 platforms implementing POSIX threads. To specify how ForeC threads are deployed on the architecture, the programmer provides a static *core-to-thread* mapping. The term *core* can designate a physical core (e.g., of the Microblaze architecture), or physical or logical threads (for PTARM and x86).

One contribution of this paper is to provide support for the FlexPRET multi-core architecture we have developed. To avoid any confusion, the term *core* designates a physical core in the context of FlexPRET and a ForeC thread is implemented by a hardware thread of FlexPRET. Section V will give examples of ForeC code executed on the FlexPRET platform.

Running Example: The TwIRTee Autonomous Robot

Our deterministic execution platform and programming language have been exercised on TwIRTee, a robotic demonstrator developed at IRT Saint-Exupery. Its primary function is to guide visitors from the entry desk to the office of the visited person or some meeting room. Given the map of the building and a goal specified as a set of way points, the mission of TwIRTee is to navigate autonomously from its initial location to its target location. To achieve this mission, TwIRTee performs three main tasks: positioning, set point generation, and tracking. Periodically, the robot calculates its position (x,y,θ), computes its next target position (called *set point*), and elaborates the linear and rotational speed commands for the wheels to track the target position.

Fig. 1 shows a ForeC model of the three threads realizing the three periodic tasks. Given the one-to-one correspondence between *ForeC thread* and *task*, we will employ these two terms indifferently in the rest of the paper. All the tasks are programmed in a *single* ForeC program. Hence, data and control dependencies between tasks are managed by ForeC directly. Input variables are shown on the left side of the figure and output variables on the right side. The ForeC program consists of a main function (application's entry point), which is composed of three threads *Positioning*, *SetPoint Generation*, and *Tracking*. A shared variable called *position* is emitted at every global tick by the *Positioning* thread and consumed by the two others. A second shared variable called *set point* is emitted by *SetPoint Generation* and consumed by *Tracking*.

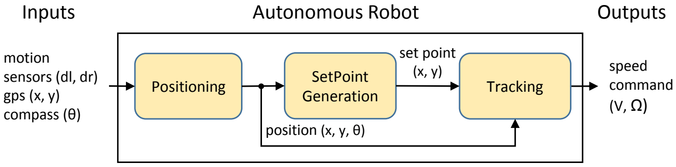


Fig. 1: TwIRTee model.

```
 1 input   Position gps;    // gps position
 2 input   int dl, dr,      // motion sensors
 3             theta;       // compass
 4 output int v, w;         // linear/rotational speed
 5
 6 Map map = ...;           // Map of the building
 7 Path path = ...;         // Mission to follow
 8
 9 void main () {           // Application's entry point
10   shared Position p;     // Current position
11   shared Position sp;    // Seed set point
12   par(
13     Positioning(dl, dr, theta, gps, p),
14     SetPointGen(p, sp, map, path),
15     Tracking(p, sp, v, w));                  }
16
17 thread Positioning(in int dl, in int dr,
18                    in int theta, in Position gps,
19                    out Position p) {
20   while (1) {
21     // Calculate position from various sources
22     p = kalman(dl, dr, theta, gps);
23     pause; /* end of local tick */           } }
24
25 thread SetPointGen(in Position p, out Position sp,
26                    Map map, Path path) {
27   while (1) {
28     // Get the projection on the path
29     Point h = projection(p, path);
30     // Compute the next target coordinate
31     sp = getSetPoint(h, path);
32     pause; /* end of local tick */           } }
33
34 thread Tracking(in Position p, in Position sp,
35                 out int v, out int w) {
36   while (1) {
37     // Kanayama call to track the defined set point
38     kanayama(p, sp, &v, &w);
39     pause; /* end of local tick */           } }
```

Listing 1: TwIRTee autonomous robot code snippet in ForeC.

Listing 1 gives an overview of the ForeC code of the three tasks that are executed in parallel (`par` statement of lines 12–15). The main function (line 9) is the application's entry point. Five input variables (lines 1–3) and two output variables (line 4) have been globally defined. Two global variables are defined for the map and the path to follow (lines 6–7). Their initialization is not described here. Two shared variables p and sp (lines 10–11) respectively contain the current position and the speed set point. We use two well known algorithms, *Kalman filtering* and *Kanayama tracking*, for calculating respectively the current position of TwIRTee based on various noised sources and the speed command. Each periodic task is executed in a loop. `Pause` statements (lines 23, 32, and 39) are used to synchronize all threads between each tick.

Fig. 2 shows a trace of the execution of the ForeC application. The rectangles show the instructions of Listing 1 that are executed. Black rectangles denote `pause` statements. Triangles denote threads that are forked by the `par` statements. Finally, the vertical axis shows the global ticks. The figure shows the first two ticks that are executed. During the first tick, the main function executes its instructions and forks the child threads. Each child thread then executes in parallel and pauses until all threads complete their processing. Once all local ticks terminate, output variables are emitted.

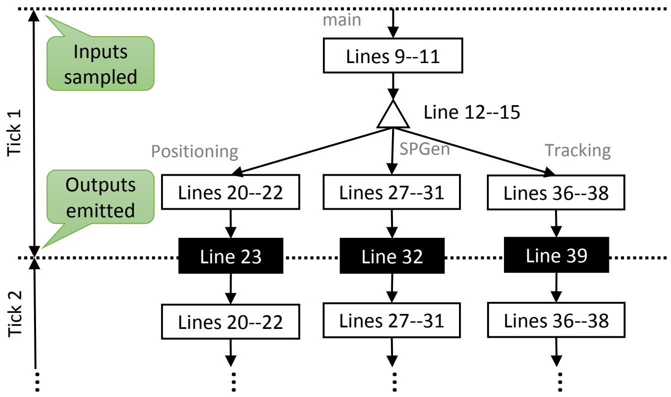
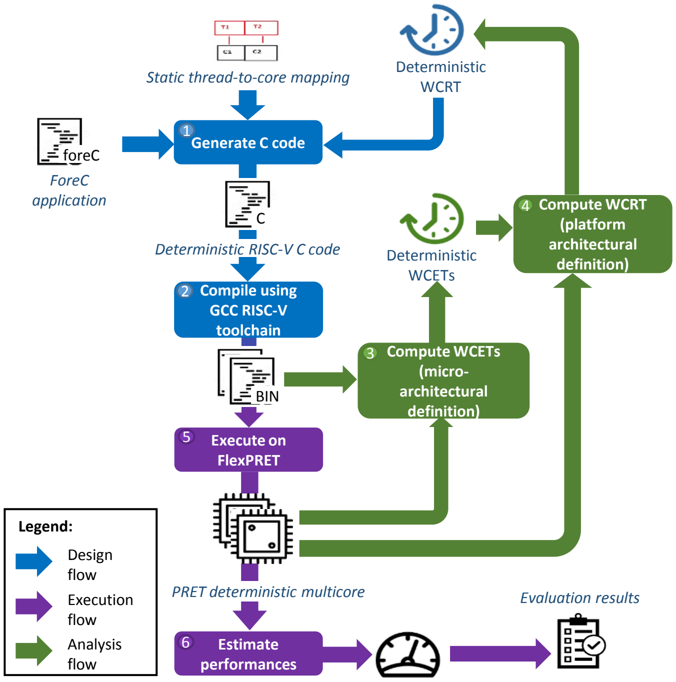


Fig. 2: ForeC execution trace.

IV. Optimization Approach

Now that the application has been programmed with ForeC, our objective is to optimize its actual implementation in order to minimize the WCRT of the main functional chain. Our approach is depicted on Fig. 3.

It consists of 6 steps. Given an application written in ForeC and a static thread-to-core mapping, a C program distributed over the different cores of the platform is generated (step 1). This program transposes the *logical timing constraints* specified by the ForeC synchronous program into *physical timing constraints* expressed using the specific timing instructions of FlexPRET (see Section V). From step 1, binaries of the distributed C program are generated for each core using the RISC-V GCC compiler (step 2). At this stage, the



T1
T2
C1
C2
Static thread-to-core mapping
Deterministic WCRT
foreC
ForeC application
1 Generate C code
C
Deterministic RISC-V C code
Deterministic WCETs
4 Compute WCRT (platform architectural definition)
2 Compile using GCC RISC-V toolchain
BIN
3 Compute WCETs (micro-architectural definition)
5 Execute on FlexPRET
PRET deterministic multicore
Evaluation results
6 Estimate performances
Legend:
Design flow
Execution flow
Analysis flow

Fig. 3: Approach overview.

program can be executed on the target FlexPRET platform (step 5), but this would violate the execution semantics of ForeC as no concurrency timing analysis has been performed. Consequently, synchronizations between tasks have not been made explicit yet. Timing analyses have to be performed such that: (i) an upper bound of the WCET is determined for each function (step 3), and (ii) the WCRT of the whole application is computed (step 4).

Computing WCETs is commonly done through measurement [28] or static analyses with, e.g., OTAWA [29]. Measurement approaches often lead to under-estimating WCETs while static analyses often lead to over-pessimistic upper bounds [30]. One of the advantages of the chosen architecture (FlexPRET) is that its micro-architecture remains simple and exempt of any source of non-determinism, which allows precise WCETs to be computed[3][4] or measured. Besides, ForeC generates bare metal code, which facilitates the inclusion of context switch costs in the computation/measurement of the WCETs. Our approach is independent of the chosen technique, but static analysis requires the hardware to be accurately modeled, which is an ongoing work. Hence, for now, WCETs are only obtained through measurement.

Computing the WCRTs on the application relies on the *platform architectural* definition to access the shared memory of the target platform (number of cores, interconnect, memories) and its efficient resolution constitutes the main objective of this paper. The output of the WCRT computation step is used to update the C program to comply with the synchronous execution semantics and to provide a fair access to the memory.

Our approach relies on the time-triggered model detailed in Section IV-A. Based on the proposed model and the knowledge of the WCET of each task running on the different cores, ILP techniques are used to compute the configuration of the interconnect that will minimize the WCRT of the application. The resolution of the model is done offline at *design time* and is dependent to the design of the interconnect. We have successively designed two conventional constrained TDMA buses (with fixed-length and variable-length time slots) which guarantees exclusive access to the global memory at hardware-level and a pure software implementation of the TDMA using a unconstrained bus. Sections IV-B to IV-D successively describe the three communication architectures.

In the following, we assume that the three communication architectures are variants of the general architecture with n FlexPRET cores depicted in Fig. 4. For the sake of clarification, we will call the resulting multi-core architecture *MultiPRET* to differentiate it from the original FlexPRET architecture in the rest of this paper. Each core is identified by its index $i \in [0, n[$ and includes two on-chip scratchpad memories, respectively for instructions and data, and a conventional 5-stage pipeline where an arbitrary number of threads can be interleaved. Inter-core communication is solely achieved via a global memory accessible through an interconnect. Local (private) memories for instructions and data of a core are accessible by all threads allocated to it, while global (shared) memory for data is accessible by all threads of all cores. All the cores are driven by the same clock and start at the same time $t = 0$. Memory accesses are performed in one CPU cycle and the chosen interconnect has no added latency.

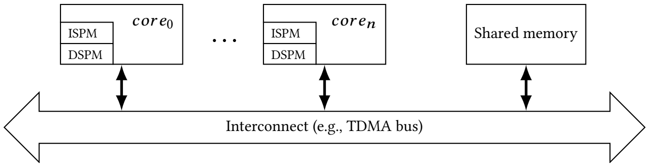


Fig. 4: MultiPRET architecture overview.

A. Time-Triggered Model

Each core can execute multiple tasks (ForeC threads) on separate hardware (FlexPRET) threads. A task executed on thread i deployed on core j is denoted by $\tau_{i,j}$. Given our synchronous model of computation, each execution of a periodic task can be decomposed into the following steps: 1) create local copies of shared variables, 2) perform application-specific operations, 3) update values of the shared variables, and 4) synchronize with other tasks executed on potentially other cores. Accesses to the shared memory are only performed during steps 1 and 3. Consequently, delays must be correctly positioned to ensure that all tasks access the shared memory during their allocated time slot. The model we propose is quite similar to existing time-triggered models in the literature, such as Acquisition Execution Restitution (AER) [32]. The originality resides in its automatic resolution through ILP and its inherent support by ForeC for automatically partitioning an application into the three phases *reading*, *operating*, and *updating* (see Section IV-B).

The period $T_{i,j}$ of a task $\tau_{i,j}$ is computed by the following ILP equation:

$$T_{i,j} = t_{d1}^{i,j} + t_{c}^{i,j} + t_{o}^{i,j} + t_{d2}^{i,j} + t_{u}^{i,j} + t_{s}^{i,j} \tag{1}$$

Where $t_{d1}^{i,j}$, $t_{c}^{i,j}$, $t_{o}^{i,j}$, $t_{d2}^{i,j}$, $t_{u}^{i,j}$, and $t_{s}^{i,j}$ are expressed in CPU cycles and respectively denote the number of required cycles to: **d**elay until the next allocated time slot ($t_{d1}^{i,j}$); perform the initial **c**opy of the shared variables ($t_{c}^{i,j}$); perform specific **o**perations of the task ($t_{o}^{i,j}$); **d**elay until the next allocated time slot ($t_{d2}^{i,j}$); **u**pdate the modified shared variables ($t_{u}^{i,j}$); and **s**ynchronise with all other tasks of the system ($t_{s}^{i,j}$). When the task $\tau_{i,j}$ is implicitly known, we will note t_{d1} to keep the notation simple, and similarly for the other delays. At this point, no assumption is made of the chosen interconnect. Allocated time slots can correspond to physical slots of a bus controller or to conceptual slots to access to the global memory in an exclusive way (as in a memory wheel).

Example 1: Let us consider the execution trace of Fig. 5 showing two tasks $\tau_{0,0}$ and $\tau_{1,1}$ respectively running on $core_0$ and $core_1$. The horizontal axis shows the CPU cycles. The two lines show the execution of the two tasks segmented with respect to Eq. (1). Each segment is represented by a rectangle. Segments during which a task is waiting for synchronization

[3] OTAWA provides a preliminary loader for the RISC-V ISA [31].

[4] WCET-aware C Compiler (WCC) for RISC-V is also to consider [30].

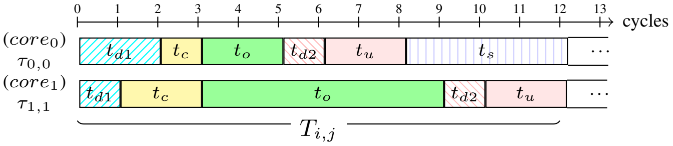


Fig. 5: Example of an execution trace with two tasks.

with its siblings or for being granted access to the global memory are hashed. Segments representing operations performed by the task are filled.

In Eq. (1), t_c, t_o, and t_u are computed using WCET computation techniques. Our goal is to properly set the values of the three synchronization delays t_{d1}, t_{d2}, and t_s for each task $\tau_{i,j}$ so that the timeliness of the execution and the absence of concurrent access to the shared memory are guaranteed while minimizing the WCRT. When not properly set, core interferences may occur. In Fig. 5, we can see that the two segments $t_c^{0,0}$ and $t_c^{1,1}$ are partially overlapping at $t = 2$, which may lead to core interferences.

Since the program is synchronous, all the tasks must share the same period T, yielding the following ILP equation:

$$T_{i,j} = T \tag{2}$$

This common period T is the *WCRT* of the application. Minimizing the WCRT is a linear optimization problem and can be solved with ILP:

$$Minimize\ T \tag{3}$$

According to Eq. (1), minimizing T amounts to minimizing t_{d1}, t_{d2}, and t_s for each task $\tau_{i,j}$.

It is notorious that solving scheduling problems using ILP may lead to scalability issues [33]. However, (i) our problem only depends on the number of cores and not on the number of threads interleaved (see below), and (ii) we only target small to medium-size multi-cores. Accordingly, we are confident that ILP is a suitable approach in this precise case, as we will show in Section V.

The resolution of Eq. (3) being dependent on the chosen interconnect, the following describes the three interconnects we implemented.

B. Fixed-Length Time Slot TDMA Bus

Our first MultiPRET architecture is similar to the one presented in [5]. It consists of n FlexPRET cores that are instantiated and connected together through a TDMA bus. The bus controller guarantees an exclusive access to the shared memory for each core based on the definition of time slots. The time slot size $t_{slot} \in \mathbb{N}$, period $T_{tdma} = t_{slot} * n$, and offset $t_{offset} \in [0, T_{tdma}[$ are configurable. In our first architecture, slots have constant sizes. A null initial offset means that the first slot opens for $core_0$ at $t = 0$.

$t_{d1}^{i,j}$ and $t_{d2}^{i,j}$ are expressed by the following ILP equations:

$$t_{d1}^{i,j} = (k_1 * n + j) * t_{slot} - t_{offset} \tag{4}$$

$$t_{d2}^{i,j} = (k_2 * n + j) * t_{slot} - t_{offset} - t_{d1}^{i,j} - t_c^{i,j} - t_o^{i,j} \tag{5}$$

where i is the thread index, j the core index, n the number of cores, t_{slot} is a constant, t_{offset} is computed by ILP, and k_1 and k_2 are two newly introduced ILP variables to make sure that the t_c and t_u segments (resp. reading from and writing to the global memory) must start exactly when the TDMA time slot opens for $core_j$.

In the multi-threaded mode, threads allocated to the same core are interleaved, meaning that only one thread executes an instruction at the same time (in the *execution* stage of the pipeline). Thus, there is no concurrent access to the memory between threads allocated to the same core as all memory accesses are performed in one CPU cycle only. We assume that all threads allocated to the same core can access the shared memory within the same TDMA time slot. Therefore, Eq.s (4) and (5) do not depend on the number of threads interleaved and hold for both single-threaded and multi-threaded execution modes. However, interleaving multiple threads on the same core impacts the WCET for each interleaved thread. An orthogonal goal is to find the best allocation of ForeC threads to FlexPRET cores given the number of available cores and other allocation constraints. This is not addressed in this paper whose primary focus is on the optimization of the WCRT of an application composed of multiple threads whose allocation has been given.

The third synchronisation delay $t_s^{i,j}$ is required to synchronize all the tasks and to align their periodic execution (during the subsequent ticks) with the same TDMA time slot that was opened during the former tick. Ignoring it would cause a misalignment of the TDMA time slots during the subsequent ticks, hence breaking the timed-predictability of the approach. Thanks to FlexPRET [3], the CPU cycles during which the HRTTs are waiting are not wasted and can be used to execute SRTTs on the same core. $t_s^{i,j}$ is computed by the following ILP equation in such a way that the period T_τ is a multiple of T_{tdma}:

$$\begin{aligned} t_s^{i,j} \quad = \quad & (k_3 * n + j) * t_{slot} - t_{offset} \\ & - t_{d1}^{i,j} - t_c^{i,j} - t_o^{i,j} - t_{d2}^{i,j} - t_u^{i,j} \end{aligned} \tag{6}$$

where k_3 is a newly introduced ILP variable to make sure that all the $\tau_{i,j}$ tasks finish their period at the same time and remain aligned with the TDMA slots.

Computing t_{d1}, t_{d2}, and t_s for each task amounts to minimizing the values of the k_1 and k_2, and k_3 while satisfying:

$$t_{d1} \geq 0 \quad \wedge \quad t_{d2} \geq 0 \quad \wedge \quad t_s \geq 0 \tag{7}$$

Finally, to guarantee the "synchrony hypothesis", we must ensure that, for any two tasks $\tau_{i,j}$ and $\tau_{k,\ell}$, we have:

$$t_{d1}^{i,j} + t_c^{i,j} + t_o^{i,j} + t_{d2}^{i,j} \leq t_{d1}^{k,\ell} + t_c^{k,\ell} \tag{8}$$

that is, the duration required by $\tau_{i,j}$ to successively: 1) wait for its allocated time slot, 2) create local copies of shared variables, 3) perform its main operations, and 4) wait for its next allocated time slot, allowing it to propagate the modifications made to the shared variables cannot be less than the duration required by $\tau_{k,\ell}$ to create local copies of shared variables. Violating this property could cause a thread to read

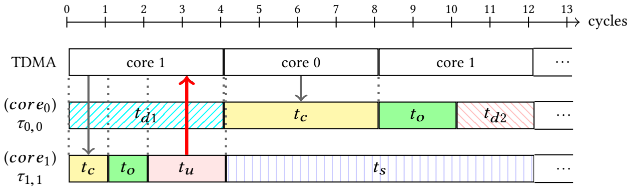


Fig. 6: Execution trace violating Eq. (8).

updated values of shared variables that would normally be visible only within the next global tick.

To illustrate Eq. (8), let us consider the execution trace depicted by Fig. 6. The first line shows the TDMA time slots. Fig. 6 shows that the task $\tau_{1,1}$ executes faster than $\tau_{0,0}$, making it possible for $\tau_{1,1}$ to update (t_u of $core_1$) the values of the shared variables before they are copied (t_c of $core_0$) by $\tau_{0,0}$. Eq. (8) guarantees that such scenarios never occur.

Overall, the ILP problem is to minimize Eq. (3) under the constraints of Eqs. (1)–(2) and Eqs. (4)–(8). Recall that the TDMA period T_{tdma} and the TDMA time slot size t_{slot} are constants of the ILP solver, i.e., they are provided by the system's designer. In the first architecture, t_{slot} is set to the largest access a task can require. This design choice ensures that a read or write operation from/to the shared memory fits inside a single TDMA time slot. A possible improvement will be to segment long read or write operations into several parts, each one fitting in a TDMA slot. T_{tdma} is set to the time slot size t_{slot} multiplied by the number of cores n.

Example 2: let us consider the three tasks of our running example allocated on three different cores. In this example, each core executes exactly one thread in single-threaded mode. So we simplify the notation by labelling the three tasks τ_{pos}, τ_{sp}, and τ_{track} and we compute their corresponding period to minimise $T_{\tau_{pos}}$, $T_{\tau_{sp}}$, and $T_{\tau_{track}}$. All cores are configured as single-threaded, so all threads execute during each CPU clock.

Tab. I gives the details of the computation of the period of each of the three tasks. The different values of the execution times t_c, t_o, and t_u for each task are estimated on the left side. Those values can be obtained through timing analyses or measurement. Based on the constants provided in Tab. Ia, we compute the variables provided in Tab. Ib using the GLPK ILP solver. The optimum of the global period T is found for an initial offset of 10 given a time slot of size 5.

Fig. 7 shows a trace of the execution of the three tasks τ_{pos}, τ_{sp}, and τ_{track} respectively allocated to $core_0$, $core_1$, and $core_2$. The execution has been configured with respect to the ILP solution given in Tab. I. We consider that the periodic tasks all start at $t = 0$ and that the TDMA bus is already configured at this time. This is of course a simplification since τ_{pos} must configure the TDMA slot before executing its periodic task. Given an offset of 10, the first TDMA time slot is for $core_2$. Task τ_{track} has no waiting segment for reading from the shared memory, since the first TDMA time slot is opened for $core_2$.

TABLE I: Period computation (fixed-length slots).

(a) ILP constants

WCETs	t_c	t_o	t_u
τ_{pos}	4	4	4
τ_{sp}	5	7	2
τ_{track}	2	6	4

t_{slot}	5
T_{tdma}	15

(b) ILP variables

Delays	t_{d1}	t_{d2}	t_s
τ_{pos}	7	5	6
τ_{sp}	10	3	3
τ_{track}	0	7	11

t_{offset}	10
T	30

C. Variable-Length Time Slot TDMA Bus

In contrast with the first architecture (Section IV-B), the size of the time slots allocated to different cores may be different for each core.

We denote $t_{slot_j} \in \mathbb{N}$ the time during which a TDMA slot is opened for core j to create local copies of the shared variables manipulated by the threads deployed on that core. Similarly, we denote $t'_{slot_j} \in \mathbb{N}$ the time during which a TDMA slot is opened for core j to update the local copies modified by the threads deployed on that core. In comparison to the first architecture where the period of each task was a multiple of the period of the TDMA, the second architecture allows for setting the period of each task identically to the period of the TDMA. The common period T is computed by the following ILP equation:

$$T = \sum_{j=0}^{n-1} (t_{slot_j} + t'_{slot_j}) \tag{9}$$

Again, solving the time-triggered model involves computing the communication delays td_1, td_2, and t_s for each core. Therefore, td_1, td_2, and t_s for each task $\tau_{i,j}$ can be computed by the following ILP equations:

$$t_{d1}^{i,j} = \sum_{k=0}^{j-1} t_{slot_k} \tag{10}$$

$$t_{d2}^{i,j} = \sum_{k=0}^{n-1} t_{slot_k} + \sum_{k=0}^{j-1} t'_{slot_k} - t_{d1}^{i,j} - t_c^{i,j} - t_o^{i,j} \tag{11}$$

$$t_s^{i,j} = \sum_{k=0}^{n-1} t_{slot_k} + \sum_{k=0}^{n-1} t'_{slot_k} - t_{d1}^{i,j} - t_c^{i,j} - t_o^{i,j} - t_u^{i,j} - t_{d2}^{i,j} \tag{12}$$

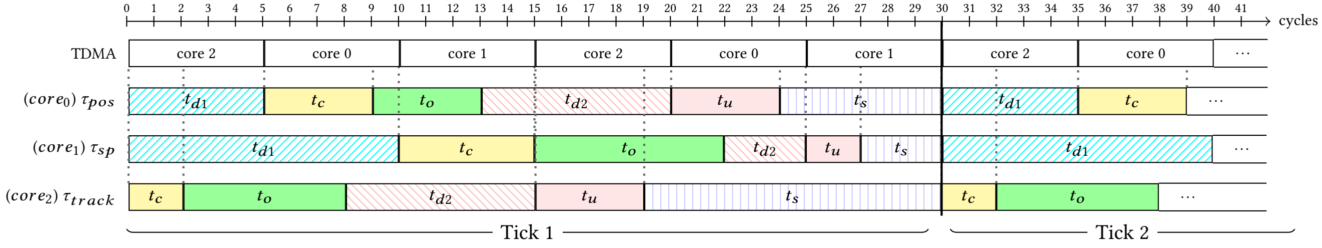


Fig. 7: Execution trace with fixed-length TDMA slots (solution 1).

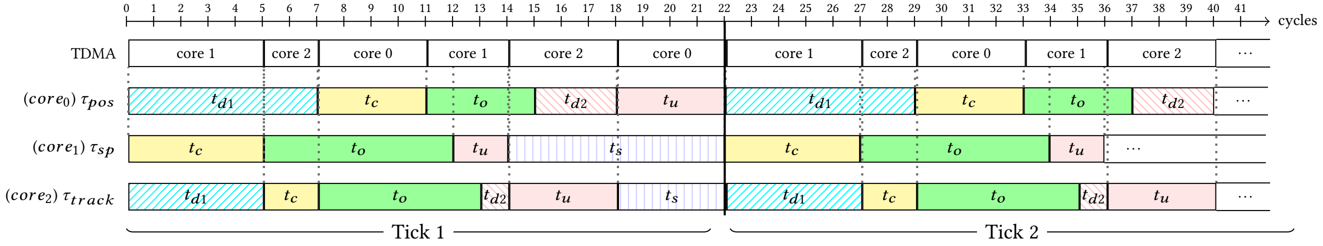


Fig. 8: Execution trace with variable-length TDMA slots (solution 2).

Eq. (10) means that a task deployed on core j has to wait until its first time slot is opened, meaning the time slot for all cores k (with $k < j$) has ended, assuming the time slots are opened in a round robin manner from the smallest index to the biggest. Tasks on $core_0$ being the first tasks to have access to their time slot, td_1 is null. Similarly, Eq. (11) states that to be able to update local copies of shared variables, the same task has to wait until all tasks have had access to their initial time slot (for creating local copies of shared variables) and that the second time slot (for updating local copies of shared variables) of all tasks on cores k (with $k < j$) has ended. Finally, Eq. (12) means that t_s must be computed in such a way that it represents the remaining time after all time slots have been opened and all operations for core j have been done. Unlike the previous section where t_{slot} was a constant of the ILP model, resolving this model implies computing each t_{slot_k} and t'_{slot_k} such that the global TDMA period T_{tdma} is minimized.

Overall, the ILP problem is to minimize Eq. (3) under the constraints of Eqs. (1)–(2) and Eqs. (8)–(12).

TABLE II: Period computation (variable-length slots).

	t_{d1}	t_{d2}	t_s
τ_{pos}	7	4	0
τ_{sp}	0	0	8
τ_{track}	5	1	4

t_{slot_0}	4	t'_{slot_0}	4
t_{slot_1}	5	t'_{slot_1}	3
t_{slot_2}	2	t'_{slot_2}	4
t_{offset}	4		

T	22

Example 3: let us consider the same example where the three tasks are deployed on three separate cores. Tab. II shows the ILP variables computed by our second ILP solver based on the WCETs calculated in the previous example (cf. Tab. Ia). We can observe that optimizing the size of the TDMA time slots individually for each core reduces the size of the tasks' period to 22 cycles, hence a decrease of 26.67% of the common period T. The optimum is found for a TDMA offset of 4, aligning the first time slot with $core_1$. To find this optimum, three ILP resolutions needed to be performed, to test for which core the first TDMA slot needs to be opened. Fig. 8 shows the resulting execution trace.

D. Pure Software Implementation of the TDMA

Traditionally, TDMA buses provide hardware mechanisms to ensure that concurrent accesses cannot occur. If two cores attempt to access to the shared memory at the same time, only the access from the core to which a TDMA time slot is opened will succeed. The combination of the ForeC synchronous language and the MultiPRET deterministic architecture allows us to dispose of such hardware mechanisms. Indeed, the time-triggered model we propose ensures *by construction* that communication for a core only occurs when the time slot is opened for this core.

To push the idea further, we propose a third architecture relying on a simple unconstrained bus connecting all cores to the shared memory and on a pure software implementation of the time-triggered model. The unconstrained bus acts as a multiplexer and does not guarantee the absence of concurrent accesses since this is already guaranteed at software-level. This frees ourselves from ensuring it at hardware-level, reducing the complexity of the hardware implementation. The unconstrained bus does not lead to a different ILP model per se; rather, it can be used in conjunction with any of the two previous ILP models.

V. Evaluation

We have extended FlexPRET to support the instantiation of multiple cores and we implemented the two software-managed TDMA bus architectures and the unconstrained bus in Chisel [34]. Our extension is called MultiPRET. We extended the memory hierarchy of [3]. A memory access at an address below `0x40000000` results in an operation to a local memory. A memory access at an address above `0x40000030` results in an operation to the shared memory. All memories are implemented using SPMs and are accessible in one CPU cycle. This is ensured by our hardware implementation onto the FPGA. Addresses in the range `0x4000000C`–`0x40000030` are reserved to configure the fixed-length and variable-length TDMA buses.

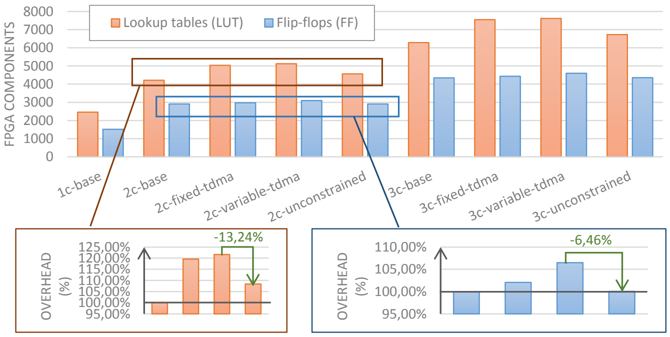


Fig. 9: FPGA resources.

A. *FPGA Resources*

We have synthesised several variants of MultiPRET on a Zynq UltraScale+ ZCU102 board. SPMs are implemented using block RAMs. Fig. 9 lists the number of Flip-Flops (FFs) and LookUp-Tables (LUTs) used for the different variants of MultiPret that we have implemented. The three variants `1c-base`, `2c-base`, and `3c-base` are baseline versions used to compute the hardware overhead. `1c-base` is the original FlexPRET architecture [3] including timing instructions, with 4 threads, and 16kBytes for D-SPM and I-SPM. `2c-base` and `3c-base` feature respectively two and three cores, with no interconnect. The six other variants feature respectively two and three cores with the three proposed communication architectures. Fig. 9 shows that the unconstrained bus consumes less resources (13.24% less LUTs and 6.46% less FFs for the dual-core version) than a full-fledge TDMA bus with variable-length time slots.

B. *ForeC to C Translation*

MultiPRET can be programmed either in ForeC or in C directly. Programming in C requires the programmer to respect the time-triggered communication model by *manually* positioning delay instructions, as it is done in [8]. This is difficult and error-prone. In this section, we present a translation of ForeC into standard C for an execution on MultiPRET, so that those delays are *automatically* positioned. The translation itself is for now manual; its implementation is ongoing.

```
1  // timer constant declarations
2  const int td1Timer = 60000; /* etc. */
3  // shared input address variables
4  Position volatile* const pAddr  = (Position*) 0
       x40000048;
5  Position volatile* const spAddr = (Position*) 0
       x40000054;
6  // shared output address variables
7  volatile float* vAddr = (float*) 0x40000060;
8  volatile float* wAddr = (float*) 0x40000064;
9
10 int main() { // Tracking thread
11   // shared input variables
12   Position p, sp;
13   // shared output variables
14   float v = 0.0f, w = 0.0f;
15   // to synchronize all threads
16   delay_until_periodic(&time, initialTimer);
17   while (1) {
18     // Wait for next time slot opening
19     delay_until_periodic(&time, td1Timer);
20     // Creation of local copies of shared variables
21     p  = *pAddr; sp = *spAddr;
22     // Kanayama call to track the defined set point
23     kanayama(p, sp, &v, &w);
24     // Wait for next time slot opening
25     delay_until_periodic(&time,
26                   copyTimer + opTimer + td2Timer);
27     // Propagation of updated variables
28     *vAddr = v; *wAddr = w;
29     // Wait for synchronization among the cores
30     delay_until_periodic(&time,
31                   updateTimer + synchroTimer); }  }
```

Listing 2: Snippet of the resulting code on FlexPRET.

Listing 1 shows the translation of the TwIRTee program into C code. The full program is 4 539 Lines of Code (LoCs). Listing 2 shows a code snippet for the *Tracking* thread: the four shared variables (lines 3–8) are stored in the shared memory at addresses `0x40000048` to `0x40000064`, and local copies are created and stored into the shared memory (lines 11–14). Listing 2 does not show how shared variables are initialized into the shared memory, nor how the TDMA bus (for the fixed- and variable-length implementations) is first configured. Those are done by the first thread on the first core since initially, the TDMA time slot is opened for that thread.

Line 16 delays the execution of the periodic execution loop so that each core starts at the same time. After this delay, the periodic execution loop is activated (lines 17–31). In this loop, each thread's body is composed of three different segments: copying the input shared variables, performing the thread main operations, and updating the output shared variables that are modified. Segments are interleaved with delays according to the time-triggered model presented in Section IV. They comprise WCETs calculated for each section (copy, perform, update) to which are added the communication delays td_1, td_2, and t_s (as formulated in Section IV) required to guarantee the correct synchronization between all threads and the exclusive accesses to the shared memory. Delays are declared as constants at the top of Listing 2 (line 2), so the time to elapse within each delay instruction is computed at compile-time. Hence, all delay instructions take the exact same number of cycles to execute, which is considered and comprised within the computed WCETs. The ILP solver to calculate these delays has been implemented in CPLEX [35] and run using GLPK.

C. *WCRT Optimization and Execution*

Given a list of potential thread-to-core mapping and platform topology candidates, we now evaluate our WCRT optimization approach. We use the TwIRTee demonstrator described in Section III consisting of three tasks that can be deployed on up to three cores. Four possible mappings of the application are therefore possible: each thread is separately mapped to a separate core (multi-core, single-threaded mode), and two (out of three) threads are interleaved on a first core while the third thread is mapped to a second core (multi-core, multi-threaded modes). We have compared the four mappings on the fixed-length time slots and variable-length time slots topologies (leading to eight configurations) with a base reference where all threads are mapped to the same core (single-core mode).

1) WCET Measurement: We first measure the WCETs of the individual tasks and we compute the values of the communication delays td_1, td_2, and t_s. The experimental protocol we followed to accurately measure the different WCETs consisted in executing our three tasks (τ_{pos}, τ_{sp}, and τ_{track}) in isolation on three different cores of our tri-core implementation of MultiPRET while varying the frequency of activation F for each thread. $F = 1/1$ corresponds to the mode where a single thread is executed every cycle while $F = 1/2$ and $F = 1/3$ respectively simulate when a thread is interleaved with respectively one or two other threads. Relying on these three sets of measures to estimate the WCETs for the four

TABLE III: Measured WCETs.

	$F = 1/1$			$F = 1/2$			$F = 1/3$		
	t_c	t_o	t_u	t_c	t_o	t_u	t_c	t_o	t_u
τ_{pos}	6000	30000	1000	12000	60000	2000	18000	90000	3000
τ_{sp}	1200	180000	1000	2400	270000	2000	3600	360000	3000
τ_{track}	1200	300000	1000	2400	460000	2000	3600	580000	3000

different mappings (alongwith the base reference), hence, the eight different configurations, is possible since FlexPRET (and consequently MultiPRET) keeps *by design* a strict temporal and spatial isolation between threads [3]. Tab. III details the different values we measured for the three phases t_c, t_o, and t_u of the execution of each task. Non-proportionality to F results from the thread latency from control hazards being hidden when interleaving multiple threads [3].

2) WCRT Computation: The resulting C program is then compiled using GCC RISC-V. Simulation is performed using a C++ cycle-accurate simulator and a testbench that we have extended for our needs. Extensions include the support for multiple cores to execute in parallel and the connection to the MORSE simulation environment. The testbench, the cycle-accurate simulator, and the MORSE simulation environment execute in a closed-loop where input variables are generated by MORSE, sampled by the simulator of MultiPRET, and output variables are used to update the animation within MORSE.

Eight ILP models have been generated and resolved in order to cover the combination of the four different mappings with the two TDMA versions. Fig. 10 shows the WCRT (in μs) and its optimization compared to the base reference (in %). The horizontal axis shows the different configurations for the fixed-length (✳) and the variable-length (▲) TDMA versions. It shows that the tri-core version (configuration 1) reduces the WCRT by respectively 48.1% (fixed-length TDMA) and 51.9% (variable-length TDMA) compared to the base reference. Fig. 11 shows the core activities for the different configurations (for the variable-length TDMA version only). For the first configuration, the core executing the longest task (core 2) is efficiently utilizing all resources while the first core is idle most of the time. Both Figures 10 and 11 highlight how our approach can be used as a guideline for selecting the most efficient thread-to-core mappings depending on the desired optimization criteria. The first configuration provides the best WCRT optimization, but it also leads to unbalanced resource utilization over the cores. Alternatively, the second configuration allows for a better utilization of the resources with a gain up to 26.3% of the WCRT while using only two cores to execute the application, hence reducing the SWaP of the platform [1].

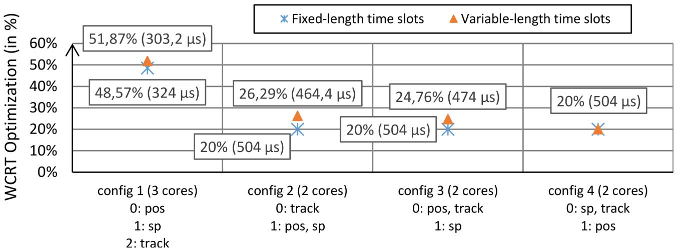


Fig. 10: WCRT optimization.

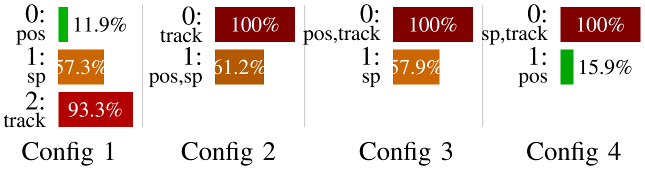


Fig. 11: Core activities (variable-length TDMA version).

VI. Conclusion

In this paper, we have combined a highly deterministic multi-core architecture, a synchronous programming language, and a communication architecture to ensure the timeliness and optimality of critical real-time applications. It is based on a synchronous model of communication in which periodic tasks are executed and synchronised at precise points in time, facilitating temporal analyses. Based on the time-triggered model and thanks to the temporal characteristics of the underlying platform (MultiPRET), applications can be written using ForeC – abstracting away temporal considerations from the programmer's point of view – and translated into standard C where communication delays are automatically calculated based on WCET and WCRT analyses. We have used an ILP solver to calculate the delays based on the platform architectural definition in order to minimize the WCRT of the application.

We have proposed three different implementations of a deterministic shared memory inter-core communication on which our time-triggered model has been applied: two constrained TDMA buses with fixed- and variable-length time slots and a simple unconstrained bus. Robustness to scheduling errors is a main difference between the three interconnects. The unconstrained bus improves WCRT and decreases FPGA footprint (13.24% less LUTs and 6.46% less FFs), but is less robust to scheduling errors than the two other interconnects. Thankfully, the time-triggered model and FlexPRET together guarantee by construction that all accesses to the shared memory are exclusive, making the hardware mechanisms provided by the TDMA buses unnecessary and disposable.

We have addressed the problem of WCRT optimization, taking into account the communication topology (physical or conceptual TDMA arbitration in this paper) and the assumption that the thread-to-core mapping is given. As future work, we plan to consider them as parts of the global optimization solution. This includes other communication topologies (partial crossbars, ring topology, . . .) and efficient mapping strategies. More generally, thanks to the flexibility of FPGAs, the communication topology can also be adapted/optimized to the application structure. Finding not only the most efficient mapping, but also the most efficient communication topology given an application profile is an interesting topic and an extension to our current work. At the same time, we are currently exploring the support of multi-rates in ForeC and its impact on the time-triggered model we have proposed.

Acknowledgment

This work is performed in the CAPHCA project of the French Institute of Technology (IRT) Saint Exupry. It is funded by the French Research Agency (ANR) and by the industrial partners of the IRT Scientific Cooperation Foundation (FCS).

References

[1] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability Considerations in the Design of Multi-core Embedded Systems," *Proceedings of Embedded Real Time Software and Systems*, pp. 36–42, 2010.

[2] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*. IEEE, 2010, pp. 339–349.

[3] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 101–110.

[4] M. P. Zimmer, "Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O," 2015.

[5] E. Yip, P. S. Roop, M. Biglari-Abhari, and A. Girault, "Programming and Timing Analysis of Parallel Programs on Multicores," in *2013 13th International Conference on Application of Concurrency to System Design*. IEEE, 2013, pp. 160–169.

[6] E. Yip, A. Girault, P. S. Roop, and M. Biglari-Abhari, "The forec synchronous deterministic parallel programming language for multicores," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2016, pp. 297–304.

[7] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 264–265.

[8] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable Programming on a Precision Timed Architecture." ACM Press, 2008, p. 137.

[9] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance." IEEE, Sep. 2012, pp. 87–93.

[10] A. Waterman and K. Asanovi, "The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.2," DTIC Document, Tech. Rep., May 2017.

[11] Xilinx, *MicroBlaze Processor Reference Guide. Embedded Development Kit EDK 13.4*, 2012, accessed September 20, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf

[12] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A Predictable Framework for Safety-Critical Embedded Systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600–1612, July 2014.

[13] E. Wandeler and L. Thiele, "Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems," in *Asia and South Pacific Conference on Design Automation*, Jan 2006, pp. 6 pp.–.

[14] T. Carle, M. Djemal, D. Potop-Butucaru, R. De Simone, and Z. Zhang, "Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays," in *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*. IEEE, 2014, pp. 112–121.

[15] A. Hamann and R. Ernst, "TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques," in *Design, Automation and Test in Europe*, March 2005, pp. 312–317 Vol. 1.

[16] D. Potop-Butucaru, A. Azim, and S. Fischmeister, "Semantics-Preserving Implementation of Synchronous Specifications over Dynamic TDMA Distributed Architectures," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2010, pp. 199–208.

[17] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Dynamic Arbitration of Memory Requests with TDM-like Guarantees," in *10th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2017)*, December 2017.

[18] H. Kopetz, "The Time-Triggered Model of Computation," in *rtss*. IEEE, 1998, p. 168.

[19] ——, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011.

[20] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java, and Real-Time POSIX*. Pearson Education, 2001.

[21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded Control Systems Development with Giotto," in *ACM SIGPLAN Notices*, vol. 36, no. 8. ACM, 2001, pp. 64–72.

[22] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zheo, and J. Zou, "PTIDES: A Programming Model for Distributed Real-Time Embedded Systems," California Univ Berkeley Dept of Electrical Engineering and Computer Science, Tech. Rep., 2008.

[23] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[24] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[25] N. Gehani and K. Ramamritham, "Real-time Concurrent C: A Language for Programming Dynamic Real-Time Systems," *Real-Time Systems*, vol. 3, no. 4, pp. 377–405, Dec 1991.

[26] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The Synchronous Languages 12 Years Later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[27] S. Natarajan and D. Broman, "Timed C: An Extension to the C Programming Language for Real-Time Systems," in *Real-Time and Embedded Technology and Application Symposium, RTAS'18*. Porto, Portugal: IEEE, Apr. 2018.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[29] H. Herbegue, H. Cassé, M. Filali, and C. Rochange, "Hardware Architecture Specification and Constraint-Based WCET Computation," in *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2013, pp. 259–268.

[30] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-case Execution Times," *Real-Time Syst.*, vol. 46, no. 2, pp. 251–300, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1007/s11241-010-9101-x

[31] "OTAWA Loader for RISC-V Instruction Set," Dec. 2017. [Online]. Available: https://github.com/hcasse/otawa-riscv

[32] C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold, "Automated Generation of Time-predictable Executables on Multicore," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, ser. RTNS '18. New York, NY, USA: ACM, 2018, pp. 104–113. [Online]. Available: http://doi.acm.org/10.1145/3273905.3273907

[33] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. De Simone, "On the Scalability of Constraint Solving for Static/Off-line Real-Time Scheduling," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2015, pp. 108–123.

[34] H. Vo, "Hardware Construction in Chisel," Master's thesis, EECS Department, University of California, Berkeley, May 2013.

[35] "CPLEX LP File Format," Sep. 2018. [Online]. Available: http://lpsolve.sourceforge.net/5.0/CPLEX-format.htm