Code generation for multi-phase tasks on a multi-core distributed memory platform

Frédéric Fort, Julien Forget

Code generation for multi-phase tasks on a multi-core distributed memory platform

Frédéric Fort
CRIStAL
Univ. Lille, UMR 9189
Lille, F-59000, France
frederic.fort@univ-lille.fr

Julien Forget
CRIStAL
Univ. Lille, UMR 9189
Lille, F-59000, France
julien.forget@univ-lille.fr

Abstract

Ensuring temporal predictability of real-time systems on a multi-core platform is difficult, mainly due to hard to predict delays related to shared access to the main memory. Task models where computation phases and communication phases are separated (such as the PRedictable Execution Model [23]), have been proposed to both mitigate these delays and make them easier to analyze.

In this paper we present a compilation process, part of the PRELUDE compiler [20], that automatically translates a high-level synchronous dataflow system specification into a PREM-compliant C program. By automating the production of the PREM-compliant C code, low-level implementation concerns related to task communications become the responsibility of the compiler, which saves tedious and error-prone development efforts.

Index terms - Code generation, PREM, distributed memory

1 Introduction

Multi-core hardware platforms are increasingly being used for the implementation of embedded systems, due to their potential for increasing system performances. However, implementing real-time systems on such platforms remains complex, mainly because cores share access to a central memory. This leads to contentions, which cause significant execution delays that are hard to predict, because they require to finely analyse task codes, task interferences and the contention resolution mechanisms [24].

To simplify the analysis of task interferences, the PRedictable Execution Model (PREM) [23] advocates to decouple communication phases from computation phases. For instance, the AER task model [8], a declination of the

Partially funded by the French National Research Agency, Corteva project (ANR-17-CE25-0003)

PREM model, splits each task of the system into three phases. The *Acquisition* phase loads task data and instructions from the main memory into the core's local memory. Then, the *Execution* phase performs the task computations using only local memory. Finally, the *Restitution* phase copies the results of the E-phase back into the main memory, for use by other tasks. This simplifies timing analysis because: 1) communication phases are clearly identified, so the system scheduler can schedule communications [1, 14] and avoid contentions; 2) worst-case execution time analysis (WCET) of computation phases does not need to take bus contentions into account [23].

Manually implementing a PREM-compliant program is tedious, unintuitive and error-prone. Another solution is to rely on a compiler that automates phases separation. For instance, PREM-compliant compilation for the LLVM framework has been proposed in [11, 17]. Our approach also tackles PREM-compliant C code generation but starts from a higher level of abstraction than previous approaches.

Contribution We present an extension to an existing compiler, which produces PREM-compliant code. The input of the compiler is a Prelude synchronous data-flow program [20]. The output of the compiler is concurrent multi-task PREM-compliant C code. The synchronous semantics is close to the PREM model, making the translation into PREM natural. We target a multi-core platform with distributed memory: one shared main memory plus one private scratchpad memory (SPM) for each core. According to a predefined distribution of tasks onto cores, the compiler generates a separate C code for each core. The code includes mechanisms to execute tasks periodically, synchronise task communications across cores and perform data transfers between local memories and the main memory. The main advantage of our approach is to simplify the development process, by automating the translation from the high-level specification in Prelude to the low-level implementation in C. In particular, concerns related to task communications become the responsibility of the compiler.

Validation We experiment with the Rosace case study [22], running on an FPGA platform with two NIOS-II Altera processors, using the Erika Real-Time Operating System by Evidence [9]. Because we rely on a compiler to produce the C code, we can easily compare PREM and non-PREM implementations (we simply need to recompile the Prelude program with different options). Because we rely on reconfigurable hardware (the FPGA), we can easily compare architectures with scratchpad memory or with cache memory.

2 Related works

Decoupling communications from computations, so as to improve timing predictability, was first proposed in the PRedictable Execution Model (PREM) approach [23]. PREM was first designed for improving timing predictability of

I/O peripheral accesses for a single core hardware, but was then extensively studied in the context of multi-core hardware where cores contend for shared resources.

The majority of works on PREM task models concerns timing analysis. In [6], authors have demonstrated the benefit of using PREM to reduce the pessimism of WCET analysis. The problem of designing scheduling algorithms and schedulability analysis for PREM tasks has drawn a lot of attention [1–3, 5, 14, 15, 18, 25, 30–32]. Schedulability analysis is out of the scope of the present paper. Our work relies on classic partitioned Deadline-Monotonic scheduling, using semaphores to implement inter-task synchronisations related to data-communications.

Other works have focused on the implementation of PREM-compliant applications. OS-level support or hardware drivers for the execution of PREM tasks have been proposed in [7, 27–29]. Converting legacy code into PREM-compliant code is a non-trivial task, which requires a very good understanding of the code to convert. Therefore, solutions have been proposed to automate this conversion. Light-PREM [16] is a software refactoring approach to produce PREM-compliant code from legacy code, based on memory profiling tools. In [26], authors proposed a compilation technique that produces code that executes memory access phases in parallel with computation phases. The authors of [11] proposed a technique for compiling a GPU kernel into PREM-compliant code. In [17], authors present a compiler based on the LLVM infrastructure that refactors legacy code into PREM code.

Our work is orthogonal to these approaches, in that we start from a high-abstraction language that naturally fits with the hypotheses of the PREM model. The PRELUDE language [20], belongs to the Synchronous Languages family. Compilation of synchronous languages for distributed hardware platforms was studied in [4,12,13], but with a single execution thread per CPU. Compilation into multi-thread/multi-task code was proposed for PRELUDE in [20], then for control-flow synchronous languages in [33, 34] and for SCADE in [19]. Unlike PRELUDE, [19, 33, 34] do not target systems with multiple periodicity constraints. In [21], a first version of PREM-compliant code generation for PRELUDE was proposed. However, it relies on a non-preemptive schedule computed off-line, and executed in bare metal. In comparison, the present work relies on preemptive on-line scheduling using the ERIKA OS.

3 Model

3.1 Hardware model

We consider a multi-core architecture with distributed memory. Each processor $\rho_i \in \Pi$ has access to a global shared memory $\mathcal{M}_G$ and to a private memory $\mathcal{M}_i$. We assume a static allocation of code and data to SPMs. Dynamic SPM allocation, where SPM address ranges can be shared between elements with non-overlapping lifespans is not considered here (unlike e.g. [26]). Compared to

a cache-based architecture, in our case distributed memory is apparent in the program code (local memory is explicitly addressable). Thus, memory transfers between private and global memories are handled by the PRELUDE compiler. This implies more predictable memory accesses without overburdening the programmer.

3.2 Scheduling

We assume a fixed-priority partitioned scheduler (Deadline-Monotonic in our ERIKA implementation). Task partitioning and schedulability analysis are out of the scope of this paper, so priority assignments and allocation of tasks to processors are considered to be inputs of our model. We use semaphores to implement task synchronisations required for data-communications purposes. We do not allow simultaneous execution of communication and computation phases on the same processor (unlike e.g. [26]).

3.3 Prelude

PRELUDE is a synchronous data-flow programming language. In comparison to more traditional synchronous languages, it adds primitives dedicated to the specification of real-time constraints and targets compilation into multi-task code. A simple PRELUDE program, which we will use as a running example for the rest of the paper, is provided in Figure 1. The program first declares imported nodes, sensors and actuators, whose behaviour is programmed outside PRELUDE as C functions. The main node `M` details the data-flows between the previous nodes. For instance, `C` produces value `tmp`, which is used by actuator `D`. Periods are specified on the inputs of the main node, (e.g. `A` has period 5 and offset 0). The PRELUDE compiler deduces the periods of node calls from the period of their inputs (hence, the *data-flow* nature of the language). In addition, it requires the inputs and outputs of an imported node call to be *synchronous*, i.e. to have the same rate. Since the inputs/outputs of node `C` initially have different rates, we use *rate transition* operators to make them synchronous: `A/^2` produces a flow twice slower than `A`. Similarly `tmp*^2` produces a flow twice faster than `tmp`, so the compiler infers that `D` has rate $(5, 0)$.

The synchronous semantics is a natural match with the PREM model, because it assumes that programs and functions repeatedly execute the following sequence: 1) acquire all their inputs simultaneously; 2) perform computations with no side-effect on variables other than their outputs; 3) produce all their outputs simultaneously.

3.4 Task graph

The translation of a PRELUDE program into C code consists of two main steps. First, the PRELUDE program is translated into a *task graph*. Then, the task graph is translated into C code. The present work did not require any modification on the first step, the reader is referred to [10] for details.

```
imported node C(i,j: int) returns (o:int) wcet 2;
sensor A wcet 1; sensor B wcet 1; actuator D wcet 1;

node M(A: int rate (5,0); B: rate (6,0)) returns (D: int)
  var tmp;
let
  tmp=C(A/^2, B*^3/^5);
  D=tmp*^2;
tel
```

Figure 1: PRELUDE running example.

The task graph is a directed acyclic graph $(\mathcal{T}, \mathcal{D})$. Each task $\tau_i \in \mathcal{T}$ releases a sequence of non-overlapping periodic jobs with a period T_i, where τ_i^n denotes the n-th job of τ_i. Since we assume partitioned scheduling, each task $\tau_i \in \mathcal{T}$ is assigned to a processor $\rho_i \in \Pi$ and may only execute on that processor. We denote π_i for that processor.

The set $\mathcal{D}$ defines the data-dependencies between jobs, using a model derived from [10], which allows for data-dependencies between tasks of different periods. We denote $\tau_i^n \rightarrow \tau_j^m$ when $(\tau_i^n, \tau_j^m) \in \mathcal{D}$, which means that τ_i^n produces data that is used by τ_j^m. Let $H_{i,j} = lcm(T_i, T_j)$ be the hyperperiod of τ_i and τ_j. We impose that communications follow a pattern that repeats each $H_{i,j}$, which is commonly the case in real-time applications [10]. A dependency between two tasks τ_i and τ_j is defined by a set of job pairs denoted $\mathcal{D}_{i,j}$, where by definition: $\forall (\tau_i^n, \tau_j^m) \in \mathcal{D}_{i,j} : \quad n < \frac{H_{i,j}}{T_i} \wedge m < \frac{H_{i,j}}{T_j}$. Then, we obtain $\mathcal{D}$ by unrolling all $\mathcal{D}_{i,j}$ across multiple hyperperiods:

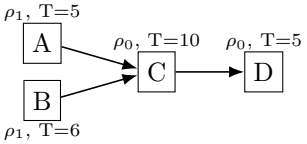
$$\mathcal{D} = \{(\tau_i^p, \tau_j^q) \mid \exists k \in \mathbb{N}, \tau_i, \tau_j \in \mathcal{T}, (\tau_i^n, \tau_j^m) \in \mathcal{D}_{i,j},$$
$$(p, q) = (n, m) + (k\frac{H_{i,j}}{T_i}, k\frac{H_{i,j}}{T_j})\}$$

The task graph for our running example is depicted Figure 2. Each two successive jobs of τ_D depend on the same job of τ_C. One out of two successive values produced by task τ_A is used by task τ_C. τ_B and τ_C have non-harmonic periods: τ_C^0, τ_C^1, τ_C^2 depend on data produced by respectively τ_B^0, τ_B^1 and τ_B^3; this pattern repeats every $H_{B,C}$.

4 Multi-phase communications

In this section, we focus on task data-dependencies, detail their semantics, and how they translate into the PREM model. In this paper, we use the conventions of the AER task model [8], a declination of the PREM model.

We assume that task communications follow a *causal* semantics. For a given execution schedule, let $begin(\tau_i^n)$ denote the date τ_i^n starts executing and $end(\tau_i^n)$ denote the date at which it completes in this schedule. Causal communication semantics impose that:



$$\mathcal{D}_{A,C} = \{(\tau_A^0, \tau_C^0)\}$$
$$\mathcal{D}_{B,C} = \{(\tau_B^0, \tau_C^0), (\tau_B^1, \tau_C^1), (\tau_B^3, \tau_C^2)\}$$
$$\mathcal{D}_{C,D} = \{(\tau_C^0, \tau_D^0), (\tau_C^0, \tau_D^1)\}$$

Figure 2: Running example

Phase	Dependency	Phase	Dependency
E_A		R_A	$\mathcal{D}_{A,C}$
E_B		R_B	$\mathcal{D}_{B,C}$
A_C	$\mathcal{D}_{B,C}$, $\mathcal{D}_{A,C}$	E_C	$\mathcal{D}_{C,D}$
E_D	$\mathcal{D}_{C,D}$		

Table 1: Phases and related data-dependencies

- Any job τ_i^n produces all its output data at $end(\tau_i^n)$;
- Any job τ_i^n acquires all its input data at $begin(\tau_i^n)$;
- For all $\tau_i^n \rightarrow \tau_j^m$, we must have: $end(\tau_i^n) < begin(\tau_j^m)$

4.1 AER phases

Let us now detail how causal communications translate into the AER model of [8]. Each task τ_i is divided into three phases. During the *Acquisition* phase (A_i), data is copied from $\mathcal{M}_G$ into $\mathcal{M}_i$. The *Execution* phase (E_i) then executes using only $\mathcal{M}_i$. Finally, in the *Restitution* phase (R_i), the results of the Execution phase are copied back from $\mathcal{M}_i$ into $\mathcal{M}_G$.

In our implementation, not all tasks have A- and E- and R-phases. First, tasks without any incoming data-dependencies, have no A-phase. Tasks without outgoing data-dependencies, have no R-phase. Second, we say that a data-dependency $\tau_i^n \rightarrow \tau_j^m$ is *local* iff $\pi_i = \pi_j$, else it is *distant*. We also say that a task τ_j is *colocated* with τ_i, iff $\pi_i = \pi_j$. If a task has only local incoming data-dependencies, it has no A-phase (no need to use $\mathcal{M}_G$). The same applies for outgoing data-dependencies and R-phases.

Phases for our running example are detailed in Table 1. For instance, τ_C copies both its inputs during A_C. Since τ_C and τ_D are colocated, their data-dependencies are directly handled by E_C and E_D.

4.2 Precedence constraints

To respect the causal semantics of data-dependencies, we impose precedence constraints between phases. Some data-dependencies impose redundant precedence constraints that can safely be removed (e.g. $\tau_C^0 \rightarrow \tau_D^1$ because τ_C is twice slower than τ_D):

$$relevant(\tau_i^n \rightarrow \tau_j^m) \Leftrightarrow \nexists \tau_i^n \rightarrow \tau_j^{m'}, m' < m \wedge \nexists \tau_i^{n'} \rightarrow \tau_j^m, n < n'$$

Let $X_i^n \rightarrow Y_j^m$ denote a phase precedence constraint (which imposes that $end(X_i^n) < begin(Y_j^m)$). By definition, we have:

$$\forall A_i^n, E_i^n, R_i^n : A_i^n \rightarrow E_i^n \rightarrow R_i^n \quad (1)$$

$$\forall \tau_i^n, \tau_j^m, relevant(\tau_i^n \rightarrow \tau_j^m), \pi_i \neq \pi_j : R_i^n \rightarrow A_j^m \quad (2)$$

$$\forall \tau_i^n, \tau_j^m, relevant(\tau_i^n \rightarrow \tau_j^m), \pi_i = \pi_j : E_i^n \rightarrow E_j^m \quad (3)$$

In our running example, precedence constraints between τ_C and τ_D stem from Equation 3, and others from Equation 2.

5 AER code generation

An overview of the compilation of a PRELUDE program is provided in Figure 3. The PRELUDE program is compiled into one C file per CPU and one C file for the global memory $\mathcal{M}_G$. Each CPU code contains one function per phase allocated to that partition, and related communication and synchronisation code (see Figure 4 for instance). The $\mathcal{M}_G$ code contains data shared for inter-processor communication purposes. In addition the C application contains code not generated by PRELUDE: 1) for each task, a user-provided *imported function*, to be executed by jobs of the corresponding E-phase; 2) the OS specific code that integrates the generated files into the final application.

In ERIKA one processor assumes the *master* role, which is in charge of declaring and initialising shared data (in our case, of handling the $\mathcal{M}_G$ data). The compilation of the C code produces one binary per processor ρ_i, to be stored in $\mathcal{M}_i$, which contains the instructions and local data of ρ_i. The binary of the master processor also contains shared communication data, which the master processor copies at boot-time into $\mathcal{M}_G$.

In our running example, processor ρ_0 is the master processor. As an example, the code generated for tasks τ_A, τ_C is provided in Figure 4. It is detailed in the next sections.

5.1 Communication buffers

For each input or output of each task, the compiler allocates a *working variable* in $\mathcal{M}_i$ that is only accessed by the phases of that task (variables suffixed by `_loc`

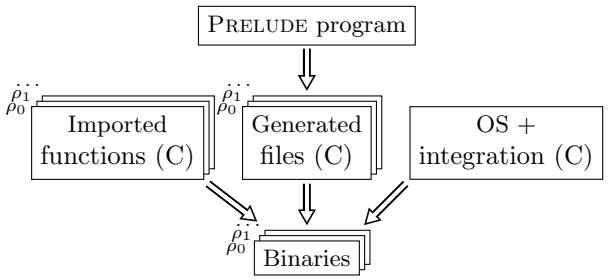


Figure 3: Overview of the PRELUDE compilation chain

```
1  // CPU 0                  // CPU 1
2  void C_A() {              void A_E() {
3    wait_sem(sem_A_C);        a_loc = A();
4    if                      }
       ↪ (must_wait_B_C())
5                            void A_R() {
       ↪ wait_sem(sem_B_C);
6                              ↪ (must_write_A_C())
7    a_loc = read_val(       write_val(
8      A_C_buff,               A_C_buff,
         ↪ A_C_idx);             ↪ a_loc);
9    b_loc = read_val(
10     B_C_buff,             if
         ↪ B_C_idx);           ↪ (must_post_A_C())
11
12   A_C_idx += 1;               ↪ post_sem(sem_A_C);
13   if                      }
       ↪ (must_change_B_C())
14     B_C_idx += 1;
15 }
16 void C_E() {
17   c_out = C(a_loc,
       ↪ b_loc);
18   C_D_buff = c_out;
19   post_sem(sem_C_D);
20 }
```

Figure 4: Generated task code of τ_A and τ_C

or `_out` in Figure 4). It allocates a *communication buffer* for each $\mathcal{D}_{i,j}$ (variables suffixed by `_buff` in Figure 4). If τ_i and τ_j are colocated, the buffer resides in $\mathcal{M}_i$ (e.g. `C_D_buff` in $\mathcal{M}_0$), otherwise it resides in $\mathcal{M}_G$ (e.g. `A_B_buff`).

In the E-phase code, the call to the imported function only operates on working variables (e.g. Figure 4 Line 17). Before this call, we must copy input data from communication buffers into working variables. After this call we must copy output data from working variables into communication buffers. For colocated communications, the copies are directly performed by the E-phase (Line 18). For distant communication, they are performed by the A/R-phases (Lines 7 Column 1, and 8 Column 2). We use the OS-specific functions `read_val` and `wrive_val` to perform copies between $\mathcal{M}_i$ and $\mathcal{M}_G$.

5.2 Multi-rate communications

The PRELUDE compiler determines for each $\mathcal{D}_{i,j}$ (see [20] for more details):

- The size of the communication buffer `i_j_buff` (e.g. `C_D_buff` is of size 2);
- A function `must_change_i_j`, which tells when to change the cell of `i_j_buff` each τ_j^m reads from (e.g. `must_change_C_D` always returns true);
- A function `must_write_i_j`, which tells for each τ_i^n if it must write in `i_j_buff` (e.g. `must_write_A_C` tells that only one out of two successive jobs of τ_A writes in the buffer);
- A function `must_wait_i_j`, which tells if τ_j^m must wait on the communication semaphore;
- A function `must_post_i_j`, which tells if τ_i^n must post on the communication semaphore.

In the ERIKA implementation, each data-dependency $\mathcal{D}_{i,j}$ is associated with a semaphore. Before copying the input for τ_j, if `must_wait_i_j` returns true we wait on the associated semaphore (Figure 4 Line 4). After copying the output for τ_i, if `must_post_i_j` returns true we post on the associated semaphore (Line 11). This ensures that phases execution respect the precedence constraints defined in Section 4.2.

6 Validation

We validate our work by implementing the ROSACE [22] case study on an FPGA board with two softcores, using the ERIKA OSEK-compliant RTOS.

6.1 Hardware platform

In order to allow the comparison between different hardware architectures, we rely on an FPGA development board, a Cyclone III by Altera with two NIOS-II softcores, depicted in Figure 5. The data and instruction *master ports* connect the processor to the *Avalon Interconnect Fabric*, a partial crossbar with a master/slave behaviour, which serves as a hub to access shared resources of the board. In our case, only NIOS-II processors are masters. Each master is only connected to a subset of slaves. Accesses from two masters to two different slaves can execute simultaneously. Simultaneous accesses to the same slave are arbitrated with a round-robin policy. *Tightly-coupled* data and instruction ports offer private contention-free access to processor-dedicated on-chip memories. Each processor has access to a tightly-coupled memory for data and to another for instructions. These memories serve as scratchpad memory ($\mathcal{M}_i$).

Table 2: Size of memories for the experiments

Memory (SPM architecture)	Size
Data SPM	ρ_0: 5kB, ρ_1: 4kB
Instruction SPM	ρ_0: 12kB, ρ_1: 8kB
Main	2kB

Memory (cache architecture)	Size
Data cache	2kB
Instruction cache	4kB
Main	29kB

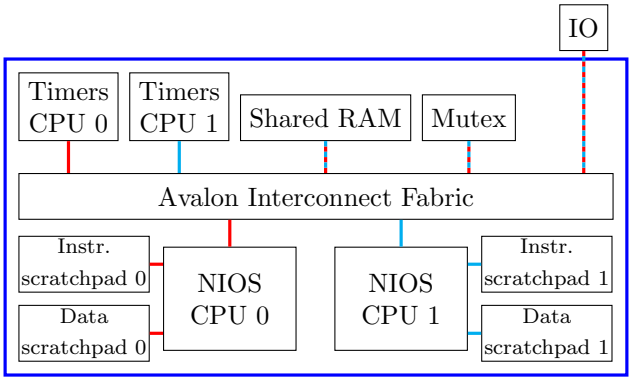


Figure 5: The hardware design.

Processors share access to an on-chip shared RAM ($\mathcal{M}_G$). On a real embedded board, the shared memory would be an external component (e.g. SDRAM, SRAM), with typically longer access time. Therefore, our shared RAM is controlled by an artificially slower clock which mimics these slower accesses. Finally, processors are also connected to an on-chip mutex, on-board IOs and timers, through the master ports. We use the mutex to implement synchronisations, because the processors do not have dedicated built-in primitives.

In addition to the scratchpad architecture we just detailed, we implement a cache-based architecture. It features a cache on each master port, with access performances similar to the scratchpads. The FPGA has tight space limitations (ERIKA is not available on more recent FPGA boards), memory sizes are reported in Table 2. Space reserved for SPM in the scratchpad-based architecture is instead reserved for the main memory in the cache-based architecture.

6.2 OSEK-compliant code

ERIKA is an OSEK-compliant RTOS, so tasks must all be declared statically in an *OIL* configuration file, which is generated by the PRELUDE compiler in our case. The OIL file is divided into several sections, in particular:

- `CPU_DATA` sections, which describe the hardware processors (identifier, source files, Hardware Abstraction Layer, stack address space, ...);
- `TASK` sections define the task set (CPU allocation, events to handle synchronisations, stack size, ...). In our case, each phase is declared as a separate `TASK`;
- `EVENT` sections which enable us to implement binary semaphores.

6.3 ROSACE case study

We use the ROSACE [22] case study, a longitudinal flight controller, to validate our work. It measures the airspeed, vertical speed and altitude of the aircraft, and controls the aircraft accordingly. We simplified parts dedicated to environment simulation (which are not meant to be embedded), so that corresponding tasks return dummy values.

The main benefit of the case study is to demonstrate that the PRELUDE compiler can automatically generate the ROSACE PREM-compliant C code for our distributed memory platform. This also enables us to compare different hardware architectures as shown in Figure 6. We use the response time of each task for the comparison. The figure shows the speedup of PREM code on the SPM-based architecture with respect to non-PREM on the cache-based architecture (e.g. speedup of 2 means that SPM+PREM is twice as fast as cache+non-PREM). We provide results for different RAM clock speeds: either the same as the global clock (red), 4 times slower (green) or 8 times slower (blue, which corresponds to observed latencies on an external SRAM on similar boards). We provide mean results for 20 executions for each configuration (variance is very low).

The observed speedup is proportional to the RAM clock. When the shared RAM is the slowest, the average speedup is 6.29 with a standard deviation of 2.19. When the shared RAM has the same clock as the global clock, the SPM implementation barely outperforms the cache one. The average speedup is 1.09 with a standard deviation of 0.31. This is likely due to the OS overheads of the PREM-compliant implementation, since each phase is implemented as a separate task.

7 Conclusion

We presented a method to translate synchronous data-flow programs into multi-task PREM-compliant C code, targeted for execution on a multi-core platform with distributed memory using an industrial RTOS. We validated our approach by implementing the ROSACE case study on an FPGA platform. We compared the performance of a scratchpad-based architecture with PREM-compliant code, with a cache-based architecture with non-PREM code. Switching between both versions, only requires to change the compilation options. Schedulability anal-

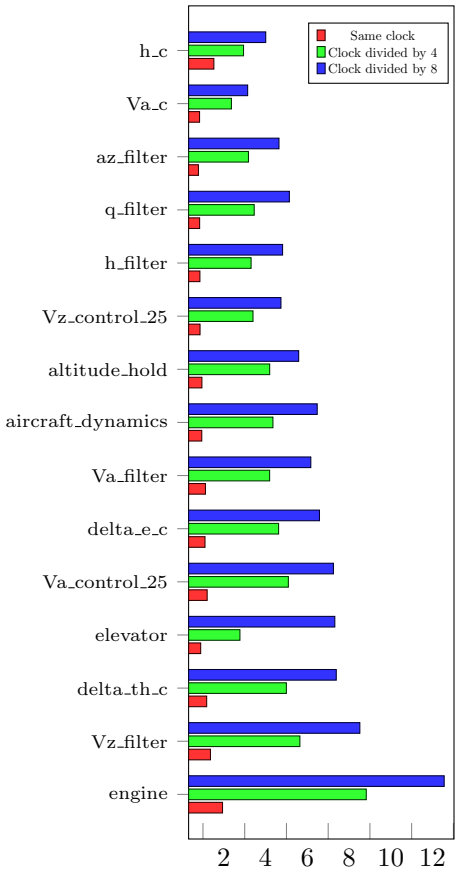


Figure 6: Observed speedup (higher is better)

ysis, which requires to consider multi-rate precedence constraints in multi-core, is left for future works.

References

[1] A. Alhammad and R. Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*. ACM, 2014.

[2] A. Alhammad and R. Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014.

[3] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015.

[4] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of signal programs. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*. IEEE, 1996.

[5] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016.

[6] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2015.

[7] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM, 2017.

[8] G. Durrieu, M. Faugere, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

[9] Erika. Erika enterprise. http://erika.tuxfamily.org/drupal/.

[10] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti. Scheduling Dependent Periodic Tasks Without Synchronization Mechanisms. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockholm, Sweden, Apr. 2010.

[11] B. O. Forsberg, L. Benini, and A. Marongiu. Heprem: Enabling predictable gpu execution on heterogeneous soc. *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[12] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(3):687–717, 2006.

[13] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*. ACM, 1999.

[14] C. Maia, G. Nelissen, L. Nogueira, L. M. Pinho, and D. G. Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017.

[15] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez. A closer look into the aer model. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016.

[16] R. Mancuso, R. Dudko, and M. Caccamo. Light-prem: Automated software refactoring for predictable execution on COTS embedded systems. In *RTCSA*. IEEE Computer Society, 2014.

[17] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu. Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2018.

[18] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *International Conference on Real Time and Networks Systems (RTNS)*, Lille, France, 2015.

[19] B. Pagano, C. Pasteur, G. Siegel, and R. Knizek. A model based safety critical flow for the aurix multi-core platform. *Proceedings ERTS2, Toulouse, France*, 2018.

[20] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[21] 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*, 2018.

[22] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The rosace case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.

[23] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011.

[24] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Transactions on Computers*, 59(3):400–415, 2010.

[25] B. Rouxel, S. Derrien, and I. Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s), Oct. 2017.

[26] M. R. Soliman and R. Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[27] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*. IEEE, 2016.

[28] R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A reliable and predictable scratchpad-centric OS for multi-core embedded systems. In *RTAS*. IEEE Computer Society, 2017.

[29] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *ECRTS*. IEEE Computer Society, 2013.

[30] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[31] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[32] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Trans. Comput.*, 65(9), Sept. 2016.

[33] 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.

[34] S. Yuan, L. H. Yoong, and P. S. Roop. Compiling esterel for multi-core execution. In *2011 14th Euromicro Conference on Digital System Design*. IEEE, 2011.