Code generation for multi-phase tasks on multicore with distributed memory

Frédéric Fort, Julien Forget

Code generation for multi-phase tasks on multicore with distributed memory*

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

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

ABSTRACT

Predicting the temporal behavior of a real-time system embedded on a multicore is a challenging task. One of the main reasons is that cores share access to the main memory, and contentions on the memory bus cause execution delays. Multi-phase task models, where computations phases and communication phases are separated (such as PREM [16] and AER [6]), 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 [15], that automatically translates a high-level system specification, consisting in a set of periodic tasks with data-dependencies, into a C program that operates according to the AER multi-phase model. The generated C program is targeted for a multicore platform with distributed memory and includes code to perform communications between the different memories of the platform.

Keywords

Code generation, multi-phase tasks, distributed memory

1. INTRODUCTION

Multicore hardware platforms are now widely used for the implementation of embedded systems, due to their potential for increasing system performances. However, the implementation of real-time systems on such a platform remains a challenge. Indeed, the increase in performance comes at the cost of more complex hardware, which implies that the timing behaviour of a program becomes more difficult to predict. One of the key factors in this complexity is memory being shared between the different cores. Contentions between cores to access the main memory cause significant execution delays. Furthermore, these delays are hard to predict, since they require to finely analyze the code of each task and interferences between the tasks.

To simplify the analyses of task interferences, the PRedictable Execution Model (PREM) [16] advocates to decouple communication phases from computation phases. The AER task model [6] follows this approach and 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 results of the E-phase back into the main memory, for use by other tasks. Such a model simplifies the timing analysis because: 1) communication phases are clearly identified, so the system scheduler can knowingly schedule communication [1, 13] and avoid contentions altogether; 2) worst-case execution time analysis of computation phases is simplified because it does not need to take bus contentions into account [16].

In this paper, we present a compilation process that generates C code compliant with the AER model. The input of the compiler is a high-level specification, in the PRELUDE [15] language, where the system is described as a set of periodic tasks with data-dependencies. The target hardware is a multicore platform with distributed memory, that is to say with one shared main memory and one local memory for each core. According to a predefined distribution of tasks onto cores, the compiler generates a separate C code for each core. The generated C code includes mechanisms to execute tasks periodically, synchronize task communications across cores and perform data transfers from local memories to the main memory. To validate our approach, we have executed the generated code on an FPGA platform with two NIOS-II Altera processors, using the ERIKA Real-Time Operating System by Evidence [7]. Our approach simplifies the development process by automating the translation from the high-level specification in PRELUDE to the low-level implementation in C. In particular, low-level implementation concerns related to task communications become the responsibility of the compiler, and can thus be handled in a more systematic and, we believe, safer way.

2. RELATED WORKS

Separating real-time tasks into phases that decouple communication from computations was proposed in the PREM approach [16] and later refined in the AER task model [6]. PREM was first designed for single core, but later extended to multicore in [19, 1, 3, 18, 13]. These works mainly focus on the problem of co-scheduling computation on the CPUs and communications on the bus. Instead, we focus on low-level implementation through automated code generation, and deliberately ignore the scheduling problem, assuming it is handled using existing techniques.

The PRELUDE language [15], which we take as input of our compilation process, belongs to the Synchronous Languages family [4]. Compilation of synchronous languages for distributed hardware platforms was studied in [2, 12, 11], but with a single execution thread per CPU. Compilation into multi-thread/multi-task code was proposed for PRELUDE in [15] and more recently for SCADE in [14], but with a single-phase task model.

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

3. MODEL

In this section, we first present the software and hardware model on which we will rely for the rest of the paper.

3.1 Task graph

PRELUDE is a synchronous data-flow programming language. In comparison with more traditional synchronous languages like e.g. LUSTRE, it adds primitives dedicated to the specification of real-time constraints and produces concurrent multi-task code instead of mono-task code. 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 (see [8] for a complete presentation), so in this paper we will consider the task graph as a starting point.

The system software is modelled as a directed acyclic task graph $(\mathcal{T}, \mathcal{P})$. Each $\tau_i \in \mathcal{T}$ is a task instantiated periodically with period $T_i \in \mathbb{N}^*$ and with relative deadline $D_i \leq T_i$.

In the AER model, each task τ_i is divided in three phases, Acquisition phase A_i, Execution phase E_i and Restitution phase R_i. During the Acquisition phase, data is copied from global memory into local memory. The Execution then performs all operations on local-memory only. Finally, in the Restitution phase the results of the Execution phase are copied back from local memory into global memory.

A directed edge $(\tau_i, \tau_j) \in \mathcal{P}$ (where $\mathcal{P} \subseteq \mathcal{T} \times \mathcal{T}$) represents a data-dependency from τ_i to τ_j. We consider *causal* data-dependencies, meaning that data-dependencies induce precedence constraints. Let P_i, P_j be two phases (either A, E or R phases), $P_i \rightarrow P_j$ denotes a precedence constraint from P_i to P_j. Data-dependencies induce the following constraints: 1) for all $(\tau_i, \tau_j) \in \mathcal{P}$ we have $R_i \rightarrow A_j$; 2) for all A_i, E_i, R_i we have $A_i \rightarrow E_i \rightarrow R_i$.

3.2 Distributed memory

For more flexibility, and in order to simplify the comparison with other hardware architectures in the future, we opted for a hardware solution that relies on an FPGA development board (Cyclone II by Altera). Our reference hardware system is depicted in Figure 1. It contains two NIOS-II CPUs, one SRAM-chip and its controller, two on-chip RAMs (scratchpads) and IOs. The SRAM and the IOs are part of the development board; everything else is directly implemented in the FPGA. NIOS-II CPUs access the SRAM concurrently, but each on-chip RAM may only be accessed by a single CPU. Local memory guarantees one 32-bit word access per clock cycle (similar to a L1 cache), while it takes 8 cycles for an SRAM access. Local memory is addressable, unlike cache memory. When the CPU emits a memory request, the Altera Avalon Interconnect dispatches the request to the correct memory controller (either local RAM or global SRAM), based on the memory address.

This kind of architecture is radically different from cache-based architectures, where cache memory is used in place of local RAM. The main difference is that in our case, distributed memory is apparent at compilation-time, i.e. in the program code. Therefore, memory transfers between global and local memories are the responsibility of the program, while they are handled automatically by cache coherency mechanisms in cache-based architectures. An important benefit of our architecture is that the cost of memory accesses is completely known statically, while it is hard to predict with cache-based architectures. This is an advantage for real-time systems, where predictability is of the utmost importance.

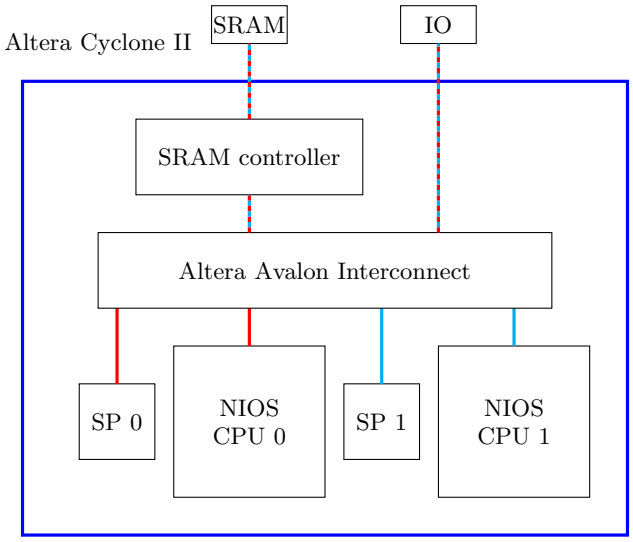

Figure 1: The hardware design

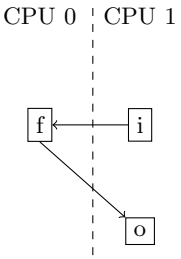

Figure 2: A simple task graph

4. CODE GENERATION WITH PRELUDE

In this section we detail the translation from a task graph into AER C code dedicated to our hardware platform. We illustrate the compilation for the simple task graph shown in Figure 2. In this example, the task i represents a sensor, task f performs logical operations and task o is an actuator. The intermediate task is located on CPU 0, while the other two are on CPU 1.

4.1 Compilation chain

The compilation of a PRELUDE program is done in two steps, as shown in Figure 3. First the high-level system specification in PRELUDE is compiled into C code. This code implements the task set corresponding to the PRELUDE program. It contains a C function for each task, communication- and precedence-related logic, along with data-structures describing task real-time properties. Then, to produce binary code, those sources are compiled along with the *imported node functions* and the *OS-specific wrapper*. Imported node functions are programmed directly in C and contain the application-specific logic. The OS-specific wrapper contains code identical for each application using a specific platform,

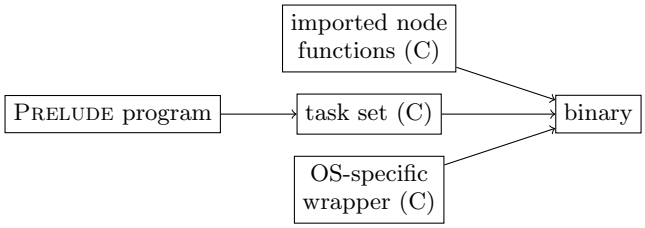

Figure 3: Overview of the PRELUDE compilation chain

```
void f_A() {
  i_f_f = copy_from_buffer(i_f_global);
}

void f_E() {
  i_f_out = f(i_f_f);
}

void f_R() {
  copy_to_buffer(f_o_global, i_f_out);
}
```

Figure 4: Illustration of the communication code

for instance the `main` function that initiates concurrent task execution. The PRELUDE distribution currently provides a wrapper for Linux, based on PTASK [17], and a wrapper for SCHEDMCORE [5]. For the present work, we added a wrapper for ERIKA.

4.2 Communications

In mono-core, PRELUDE relies on precedence encoding [9, 10] to enforce precedence constraints between tasks. However, PRELUDE programs contain extended precedence constraints (i.e. constraints between tasks of different periods), for which there currently exists no precedence encoding algorithm in multi-core. Thus, in the present work precedences are enforced using binary semaphores. Each precedence relation is associated with a semaphore. Using the information provided by PRELUDE the predecessor and the successor respectively increment and decrement the semaphore.

Communications between tasks use two kinds of buffers, global and local ones. Figure 4 shows the generated code for task f in our example program. Whenever a task (here f) wants to access the buffer between itself and its predecessor (here i), it copies during the A-phase the data from a global buffer (here `i_f_global`), into a local input buffer (here `i_f_f`). The results of the E-phase are then stored inside a local output buffer (here `i_f_out`). During the R phase, the contents of this buffer are then copied into a global buffer for each successor task (here only one successor `o`, with buffer `f_o_global`).

All memory accesses use the same addressing mechanism, the underlying hardware mechanisms automatically dispatches the requests to the right memory controller, based on the memory address. This makes the use of either kind of memory transparent. Thus, operations on either kind of memory are simply performed using a `memcpy`. However, the generated code is generic and can accept more complex copy operations for specific platforms.

CPU0	CPU1
–	`int i_f_i;`
`int i_f_f;`	–
`int i_f_out;`	–
–	`int f_o_o;`
–	`void i_E();`
–	`void i_R();`
`void f_A();`	–
`void f_E();`	–
`void f_R();`	–
–	`void o_A();`
–	`void o_E();`

Figure 5: Code distribution in our example program

4.3 Distributed code

In a NIOS-II system, each CPU runs its own binary. Thus the PRELUDE compiler produces specialized source files for each CPU, giving each CPU only the information it needs to run. Since those files use the same symbols, the wrapper of each CPU can generically configure the system at run-time.

Figure 5 shows how functions and buffers are distributed among CPUs, which is done simply by declaring them in the C file dedicated to the corresponding CPU.

4.4 OSEK compliant code

We decided to use the RTOS Erika Enterprise for our reference implementation. Being an OSEK-compliant RTOS, Erika has to follow specific rules concerning the compilation process, the structure of source files and the declaration and usage of OS primitives.

Most importantly, the OS has to be configured using a so-called OIL-file. This file specifies the entities of the system. It contains the number of CPUs, the OS tasks and the OS primitives. All tasks are defined inside the OIL-file and if they need to use a mutex or semaphore, it has to be defined there. Primitives are referenced inside the source code by the name they are given inside the OIL-file and it is not possible to create new ones during run time. This required an additional generation step inside the PRELUDE compiler.

In contrast to other RTOS, OSEK tasks are not intended as being periodic but single-shot. All logic pertaining to their periodic behaviour is supposed to be outside of the task. In addition, tasks do not accept arguments. In the other wrappers, we used the ability to pass arguments to produce a generic task function and ship it with the wrapper. Here, it was impossible to directly reuse the task function. We could have generated the same task body for each task of the task graph, but this would have dramatically increased the binary size. Thus, we decided to write a generic function `task_do` and generate task functions that just call this function with the right arguments. Figure 6 shows an example. The first two declarations (`TASK(...)`) are OSEK task declarations generated by PRELUDE. The `static` variable is needed by PRELUDE to count the number of job executions. The first argument to `task_do` is an index used to reference the correct task in local data structures. The call to `TerminateTask` is mandatory in ERIKA to signal task completion. The `task_do` function shown here, is a simplified version, but the global idea is conserved. First, we wait on all semaphores shared with our predecessors (to wait for input data). Then, we call the task function and finally we increment all semaphores shared with our successors (to signal the availability of outputs).

```
TASK(Task_A)
{
  static int i = 0;
  task_do(0, i++);
  TerminateTask();
}

TASK(Task_B)
{
  static int i = 0;
  task_do(1, i++);
  TerminateTask();
}

void task_do(int id, int i) {
  for(int j=0;j<npreds[id];++j)
    WaitSem(preds[id][j]);

  task_params[id].fun();

  for(int j=0;j<nsuccs[id];++j)
    PostSem(succs[id][j]);
}
```

Figure 6: The generated task bodies and the generic task_do function

5. CONCLUSION

We presented a method to translate a fairly classic periodic task graph model into a C program, targetted for execution on a multicore platform using an industrial RTOS. The generated code includes code to handle task communications and task synchronizations. Task execution follows the AER model, which decouples communications from computations, thus simplifying subsequent real-time analyses. Schedulability analysis, which requires to consider extended precedence constraints in multicore, is left for future works.

6. 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] 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,*, volume 1, pages 656–665. IEEE, 1996.

[3] 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 *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pages 14–24. IEEE, 2016.

[4] 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*, 91(1):64–83, 2003.

[5] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti. Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In *19th International Conference on Real-Time and Network Systems*, Nantes, France, Sept. 2011.

[6] 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.

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

[8] J. Forget. A synchronous language for critical embedded systems with multiple real-time constraints. *Ph. D. dissertation*, 2009.

[9] 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.

[10] J. Forget, E. Grolleau, C. Pagetti, and P. Richard. Dynamic Priority Scheduling of Periodic Tasks with Extended Precedences. In *IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA)*, Toulouse, France, Sept. 2011.

[11] 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.

[12] 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*, pages 74–78. ACM, 1999.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] ptask. Periodic real-time task interface to pthreads, 2013. https://github.com/glipari/ptask.

[18] 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*, pages 1–11. IEEE, 2016.

[19] 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.