Cyclic Executive for Safety-Critical Java on Chip-Multiprocessors

Anders P. Ravn
Department of Computer Science
Aalborg University
apr@cs.aau.dk

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Chip-multiprocessors offer increased processing power at a low cost. However, in order to use them for real-time systems, tasks have to be scheduled efficiently and predictably. It is well-known that finding optimal schedules is a computationally hard problem. In this paper, we present a solution, that uses model checking to find a static schedule, if one exists at all, which gives an implementation of a table driven multiprocessor scheduler. To evaluate the proposed cyclic executive for multiprocessors we have implemented it in the context of safety-critical Java on a Java processor.

1. INTRODUCTION

Cyclic executives have been used for decades to build safety-critical real-time applications, because they are simple to understand. They give a fully deterministic scheduler and avoid dynamic locking of shared variables. It is well-known that the rigid discipline of a static schedule comes at a cost in terms of restrictions on the tasks to be periodic with reasonably aligned periods and small variations in execution times. The behavior under overload conditions is rather unpredictable as well. The pros and cons are well known, and we have little to add to this debate, see eg. [10] for an illuminating discussion. In summary, a cyclic executive with a static schedule has, compared to priority based preemptive scheduling, the following advantages and disadvantages.

Advantages:

- Determinism
- Non-interference
- Predictable worst-case execution time (WCET) (e.g., caches on preemption)
- Simple context switch because it happens at predefined points of the program
- Fewer context switches because there are none caused by preemption
- Simple dispatcher

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES'10 August 19–21, 2010 Prague, Czech Republic
Copyright 2010 ACM 978-1-4503-0122-0/10/08 ...$10.00.

Disadvantages:

- Constraints on periods
- Constraints on the maximal allowed WCET
- Frame overruns are poisonous
- No easy way to implement a bandwidth server

We find that from a programmers point of view, the constraints on periods are the most problematic. Algorithms have to be artificially split into sub-tasks to enable the construction of a schedule. When using the cyclic executive scheme on a chip-multiprocessor (CMP) we would like to keep the advantages and relax some of the constraints (disadvantages). The issue of restrictions of task periods can be relaxed as longer tasks can run on their own processor core, while on another core tasks with a shorter period can be scheduled. Furthermore, the discipline of having minor frames defined by the greatest common divisor (GCD) of periods can be relaxed even on a uni-processor. It was presumably used to ease manual schedule generation or to use non-programmable, fixed rate, coarse grained real-time clocks. However, both considerations are not relevant these days.

Although dynamic scheduling offers greater flexibility, static scheduling with cyclic executives continues to exist. Even in the proposed safety-critical Java specification (SCJ) [5], the most constrained level, which should be particularly suited for certification, prescribes the use of a cyclic executive. Thus, in order to combine the processing power of a CMP with the assurances of a static scheduling discipline, we explore the options and possible pitfalls of a cyclic execution model on a shared memory CMP.

The contribution is a flexible static schedule that allows tasks to be scheduled at varying points of time, thus avoiding the straight-jacket of minor cycles. Furthermore, tasks are allowed to move between processors between executions, which allows longer running tasks to interleave with shorter ones. An observation is that the truly parallel execution relaxes some of the restrictions of the cyclic executive model, for instant pointed out in [10]; but it also introduces conflicts in access to shared data, which we solve through additional constraints on the allowable schedules.

Schedule generation is done with a model checker, because the problem is known to be NP-complete, so we may as well use tools that are built to tackle such problems. Furthermore, as demonstrated in the Times tool, a model checking engine is very useful for analyzing scheduling problems where there are additional constraints on the tasks, e.g. dependencies.

In the following, we give a short background on related work and the work on safety-critical Java, then we introduce the execution model in Section 3, followed by a description of schedule

generation in Section 4. Section 5 gives some initial results, and Section 6 describes a concrete implementation for a Java enabled CMP. The conclusion in Section 7 touches on the advantages for precise worst-case execution time analysis of statically compiled schedules.

2. RELATED WORK

Chip-multiprocessors have renewed interest in multi-processor scheduling, which has been investigated by researchers over the years. In our initial investigations, we have been assisted immensely by a recent survey by Davis and Burns [4]. It is very comprehensive and gives a taxonomy of scheduling methods, which we will use in the following sections. However, the cyclic executive computational model in general and on shared memory multiprocessors has not received much attention in the academic world. It is presumably too simple to warrant much attention; nevertheless, Baker and Shaw give a formal definition of a cyclic executive and provide implementation suggestions within Ada [2]. In the survey mentioned above, an algorithm by Horn [7] to build a schedule for a job shop problem is mentioned. It reformulates the problem as a linear programming problem which has a polynomial time complexity (cubic) in the lowest common multiple (LCM) of the task periods. This algorithm can be adapted to independent cyclic tasks with known periods and execution times. Yet, it does not seem to have found much use.

Although not explicitly called a cyclic executive, the time-triggered architecture (TTA) for distributed real-time systems [9] assumes a static, cyclic task scheduling on the connected computing nodes. The task schedule of all nodes is synchronized via a global time base established with the time-triggered communication. This schedule is generated using a heuristic algorithm [14]. Pop et. al use a simulated annealing algorithm to find a schedule when an exhaustive search is too expensive [13].

Besides using heuristics to solve the NP-hard scheduling problem, there exist approaches to use an exhaustive search for a schedule generation. Yovine and associates have used their KKONOS real-time model checker for such purposes in a similar way to ours, see for instance [1] for a recent result. Also Metzner at. al solve the task allocation problem for distributed real-time systems with a SAT solver [11]. Realistic problem sizes (several computing nodes and up to 50 tasks) can be solved within an hour. The scheduling problem is transformed into a nonlinear integer optimization problem. The bounded integer values are replaced by boolean expressions of the 2's complement representation for the SAT solver. In our approach with model checking, bounded integer variables and the notation of time is directly supported by the timed-automata model checker, UppAal.

Finally we have the Giotto framework [6] which uses static scheduling within frames. However, this is primarily to prevent I/O-jitter. In our approach, we see limitation of jitter as an additional constraint that may be added to the schedule generation algorithm.

2.1 Real-Time and Safety-Critical Java

Under the Java community process a new standard of Java for safety-critical systems evolves [5]. Knowing that safety-critical systems, and the certification effort, covers a broad range of different criticality levels, the standard defines three levels of compliance: level 0 defines a cyclic executive, level 1 a single mission under the regime of a preemptive scheduler, and level 2 supports nested missions for more dynamic systems. It is perceived that a higher level, supporting more dynamic systems, is either more expensive to certify or will be certified to a lower safety-critical level.

The Safety-Critical Java (SCJ) specification is a subset of the Real-Time Specification for Java (RTSJ) [3]. The original RTSJ defers the issue of several processors to the Java programming model for thread scheduling. The SCJ expert group has pushed the consideration of CMPs within the RTSJ. Wellings presented the first proposal to adapt the RTSJ for multi-processors [18]. In the next release of the RTSJ (JSR 282) each schedulable object can be assigned an affinity set to guide the real-time scheduler on which processor cores the thread is eligible to execute. SCJ, as it is based on the RTSJ, provides the same mechanism of affinity sets. In the current version of the SCJ specification chip-multiprocessing is only available for level 1 and 2. The expert group has decided to keep level 0 as simple as possible and to avoid the complexity of true concurrency. As already mentioned, Doug Locke compares the cyclic executive with the preemptive tasking model [10]. He argues for the preemptive tasking model even in safety-critical systems to gain more flexibility and still be predictable. Despite his strong opinion on preemptive scheduling, he suggested within the safety-critical Java expert group to use a uniprocessor cyclic executive for level 0.[1]

In summary, as the result by Horn suggests, and as the experiences with TTA and Giotto indicate, there is no reason to consider cyclic executives uninteresting or impractical. Also, modern tools like model-checkers and SAT-solvers offer opportunities that go much beyond a hand crafted minor-cycle/major-cycle schedule.

3. THE EXECUTION MODEL

The systems that we consider are in the classification of Davis and Burns [4] called homogenous systems, because we consider M identical processors.

The application consists of N periodic tasks τ_i with a period T_i and a worst-case execution time (cost) C_i. Each release k of a task can run on one of the M processor cores. The allocation a_{ik} of release k of task τ_i is

$$a_{ik} \in P = \{1 \ldots M\}$$

Note that this permits job-level migration. Disallowing migration ($a_{ik} = a_{il}$ for all k,l) is an addtional option that reduces the set of feasible schedules.

Compared to distributed or loosely coupled systems, task migration between cores on the same chip is cheap. And it offers some flexibility. A minimal example of a task set that is schedulable with task migration, but not without is as follows: Task A and B run 2 time units and have to be scheduled once every three time units. Task C needs one time unit and has to be scheduled once every 1.5 time units. The schedule is:

```
core 0: C A A
core 1: B B C
```

This is different to the Giotto [6] model of computation, where each task is bound to a host. However, Giotto's main focus is on mode switches, where mode changes are more loosely defined in SCJ.

Note that migration assumes that the processor clocks are synchronized, because unsynchronized clocks would violate period constraints of migrating tasks. However, to use scheduling constraints for task communication the same assumption is needed. For CMPs this synchronization comes for almost free, because the individual clocks can be driven from the same oscillator.

In the following, we consider a set of N periodic tasks, where each task is characterized by its period T, its deadline, D and its

[1] Although one author is member of the SCJ expert group, this paper does not reflect the current opinion of the expert group. It is intended as a base for further discussions within the group.

WCET, C. Tasks are distinguished by indexes i and j in the following, and k and l denote release instances and range over the natural numbers.

A schedule is fully defined by: 1) the start times s_{ik} of the releases of all tasks, with $k = 0,\dots,SCM_{1\le i\le N}(T_i)$, where SCM is a function that computes the smallest common multiple of its arguments; and a processor assignment a_{ik} for a task for a given release. Since the schedule is static and non-preemptive each processor has to run an assigned task to completion, thus tasks assigned to a processor cannot overlap:

$$i \neq j \wedge a_{ik} = a_{jl} \implies [s_{ik}, s_{ik} + Ci] \cap [s_{jl}, s_{jl} + C_i] = \emptyset$$

A feasible schedule will allow computations to complete within their deadlines, therefore the start times are further constrained by

$$kT_i \le s_{ik} + C_i \le kT_i + D_i$$

Release jitter for task i may be bounded by bounding $s_{i(k+1)} - s_{ik} - T_i$ for all k.

3.1 Shared Resources

Resource sharing between tasks on a uniprocessor with a cyclic executive is trivial: the whole task is a non-interruptible critical section. On a multiprocessor system this assumption does not hold anymore. There are following options:

- Use precedence constraints between tasks in generating the schedule
- Use the simple task model, *read - execute - write* , and constrain only the presumably short *read* and *write* sections.
- Use non-blocking queues between individual tasks
- Implement a transaction model, which is far from simple
- Implement dynamic locks (Java synchronized). However then the schedule is disturbed and there is not much win compared to preemptive multitasking

In our implementation, we use constraints when generating the schedule. Resources are modelled by boolean values R_v, $v \in 0,\dots,V$. When a task i is executing, it has a given set of used resources $CLAIM_i$ which does not vary between releases. Two tasks interfere if they use the same resources, e.g. one of them writes and the second either reads or writes to the same data structure. If two tasks interfere, they are not allowed to execute in parallel

$$i \neq j \wedge CLAIM_i \cap CLAIM_j \implies [s_{ik}, s_{ik} + Ci] \cap [s_{jl}, s_{jl} + C_i] = \emptyset$$

We have also included a schedule generation for the simple task model which essentially implements a readers-writers algorithm for access to the global shared resource.

4. FINDING A SCHEDULE

To find a schedule with model checking, each task is represented by an automaton. Figure 1 shows a simple version of the task model. The task model is not application specific; it represents each task of the application automaton for task τ_i is parameterized with the period T_i, the execution time C_i. and the deadline D.

The model uses two local clocks: t represents the real time elapsing during one period T of the task and r represents the execution time; both are initially 0. All tasks start at location Start. In this simple model all tasks become ready at the start of the schedule. Start offsets can be introduced in a transition from the location

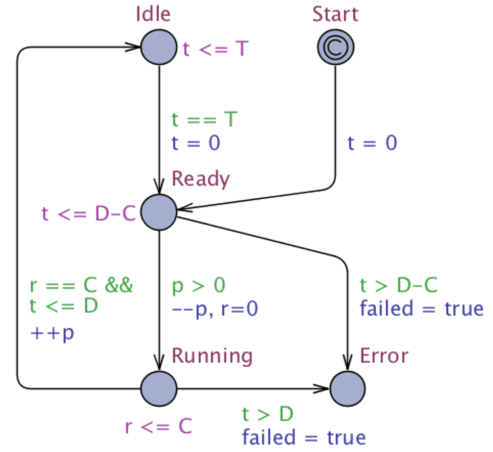

Figure 1: Simple model of a task

Start, if needed. Here, they start immediately, as Start is marked with a C for committed. It means that it will take an atomic step to the next location, Ready.

A task must transit from Ready to Running at the latest when the remaining time corresponds to the deadline minus the cost. This is given by the invariant t <= D-C. In order to leave, the task needs a processor, and as the number of free processors is modelled by a global counter p, the transition to running has the guard p > 0. If the transition is taken, p is updated and r is reset.

The task stays in state Running for the full execution time, the invariant r <= C and the guard t <= D && r == C on the transition to Idle, unless its deadline is violated, the guard t > D on the transition to Fail. In case of a deadline miss, the global variable failed is set.

If the schedule is still good, the number of free processors is increased on the transition to Idle, where the task waits for the next release, due to the invariant t <= T and the guard t == T.

When a number of task automata are put in parallel, they specify a given task set. But notice that there is no scheduler, they are free to move between Ready and Running as long as the guards and invariants are satisfied. The model checker needs this beneficial non-determinism to check whether there is a feasible schedule. With UppAal a schedule is generated by verifying that there exists a sequence of transitions (the modality E) that includes a state (the modality <>) such that a running global clock gt has advanced to the hypercycle (HC = SCM(T_i)) and failed is never set.

```
E<> gt = HC() and not failed
```

UppAal can generate a trace for one possible schedule during the verification of this property. This trace is the static schedule and can be run by the simulator or read out by other tools.

As an example for a generated schedule we use the task set of three tasks that need task migration between two processors to be schedulable. For this we set up the parameters as follows:

```
/* Processor configuration */
const int M =2; // Number of processors
typedef int[0,M] Processors;
Processors p = M; // Number of free processors

/* Task Configuration */
typedef struct{int T; int D; int C;} ReleaseParameters;

const int N = 3; // Number of tasks
typedef int[0,N-1] Id;
```

Time	Processor 1	Processor 2
0	τ_0	τ_1
1	τ_2	...
2	...	...
3	...	τ_0

Table 1: Schedule for the example task set

```
typedef  ReleaseParameters TaskSet[N];

const TaskSet TS = {
 {  2,  2,  1},
 {  4,  4,  3},
 {  4,  4,  3} };
```

The example is then instantiated by

```
Task(const Id id) = SimpleProcess(id, TS[id].T,
    TS[id].D, TS[id].C);

system Task;
```

where the system Task statement defines the network by iterating over the set of possible ID's.

The model checker can be called with several options that determines the strategy and the result. In general, for these experiments, we use a search strategy which is randomized, depth first. It will return the schedule given in Table 1. It turns out that the deadline for one, and just one, of the long running tasks can be lowered to 3, thus requiring it to run immediately when released.

4.1 Constraints on Shared Resources

Shared resources are in this model represented by a global bit vector free and each task has its bit vector CLAIM. In the automaton, the guard on the transition from Ready to Running is strengthened with (CLAIM & free) == CLAIM, which ensures that the resources are available, and the update free -= CLAIM to claim them. When leaving for Idle, they are released by the update free += CLAIM.

Adding resource constraints reduces the possible parallelism of the task set. Especially when the main execution time of a task is spent without accessing shared resources. To increase parallelism a task can be structured to represent a simple task [8]. A simple task has three phases: (1) read global state, (2) execute, and (3) write global state. If the former cyclic task is split into three subtasks, the individual tasks are less constrained relative to other tasks. The read task interferes only with tasks that write data, only the write task needs mutual exclusion. In essence, a readers-writers synchronization can be used. A simple version is shown in Figure 2

In this simple model, we assume that there is only one global resource. The number of readers is counted in rd and a write is counted in wr. There are now two additional locations P1 and P2 where a tasks is ready, but may be blocked waiting for access. The time for reding is given by a global constant RC and the time for writing is another constant WC.

The splitting of tasks into the read-execute-write subtasks can be combined with the resource control with the resource vector CLAIM. Using individual resource flags of CLAIM for the read and write tasks increases the schedulability of the tasks compared to a single global resource.

4.2 Controlling Jitter

A concern with the generated schedule could be that, the release jitter is too large. This can be controlled by constraining the running clock r <= T+J on the transition from Ready to Running. That will keep the release jitter, the time spent in Ready below a constant J except for the first release. Jitter at the first release can be controlled by adding a release offset on the transition from Start to Ready.

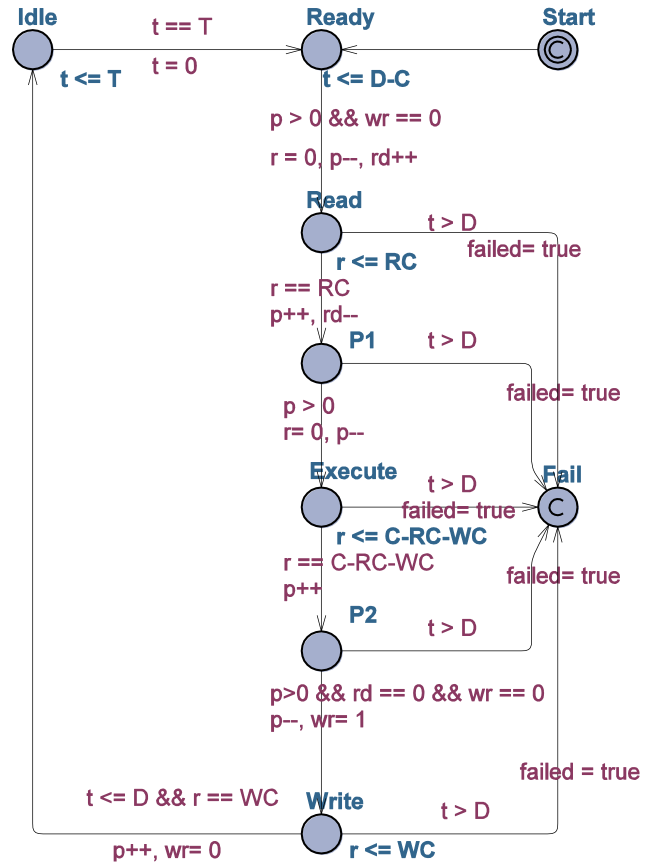

Figure 2: A task model with read, execute, and write phases

4.3 Further Constraints

The models can be further specialized in several directions. One could introduce processor pinning, by replacing the simple counter p with a bit vector analogous to the encoding of resource constraints in a claim. Task communication or event triggering can be modelled by introducing synchronization channels.

Constraints are in general beneficial, because they constrain the search space for the model checker, but they complicate applications. A further danger is that they may introduce deadlocks among the tasks. Here one could use the model checker to check for absence of deadlocks. This is not necessary for the models introduced here. They will deadlock or more precisely time-lock just when given incompatible parameter values: D < C , T < C, or T < D.

However, we will leave these modifications for future users and proceed to implementation of the executive and experiments with realistic task sets.

5. UPPAAL RUNTIME EXPERIMENTS

In order to see whether the schedule generation approach will work in practice, we have taken a 16 task workload from the real-time control of an outdoor autonomous vehicle that was developed some years ago. The parameters are as follows:

```
const TaskSet TS = {
/* T   D    C  */
{ 100, 100,  3},   // FO Gyro
{  50,  50,  2},   // Magnetometer
{1000, 200,  8},   // GPS
{ 500, 200,  6},   // Sonar
{  50,  50, 10},   // Vision
{  50,  50,  2},   // Operator Input
{ 500, 500, 10},   // Log
{  20,  20,  3},   // Supervisor
{  50,  50,  2},   // Wheel drive 1
{  50,  50,  2},   // Wheel steer
{  50,  50,  2},   // Wheel drive 2
{  50,  50,  2},   // Wheel steer
{  50,  50,  2},   // Wheel drive 3
{  50,  50,  2},   // Wheel steer
{  50,  50,  2},   // Wheel drive 4
{  50,  50,  2}    /* Wheel steer */ };
```

The utilization is 0.82 and a schedule for it is found in seconds. A variant, to check the behaviour when the minor cycle is exceeded is given by changing C = 8 to C = 17 for the GPS task. Again a feasible schedule is found in a short time. Additionally, one can do the same for the Log task, and increase from C = 10 to C = 17. A feasible schedule is found within minutes. To see, which load can be accommodated, the cost C of the supervisor is increased - for C = 6 (utilization 0.97) the task set is still feasible! But then UppAal runs for about 10 minutes on a lap-top.

In order to check a dual processor solution, we used the following modified version

```
const TaskSet TS = {
/* T   D    C  */
{ 100, 100, 3+3},   // FO Gyro
{  50,  50,  3},   // Magnetometer
{ 400, 200,  8},   // GPS was T = 1000
{ 400, 200,  6},   // Sonar was T = 500
{  50,  50, 10},   // Vision was T = 500
{  50,  50,  2},   // Operator Input
{ 400, 400, 10},   // Log
{  20,  20,  3},   // Supervisor
{  40,  40,  5},   // Wheel drive 1
{  40,  40,  3},   // Wheel steer
{  40,  40,  5},   // Wheel drive 2
{  40,  40,  3},   // Wheel steer
{  40,  40,  5},   // Wheel drive 3
{  40,  40,  3},   // Wheel steer
{  40,  40,  5},   // Wheel drive 4
{  40,  40,  3}    /* Wheel steer */ };
```

It has an utilization of 1.34. By mistake, the set was checked with one processor only, and that ran for hours without termination. This gives an indication of the huge state space that may have to be searched if there is no feasible solution. With two processors, a schedule is generated within seconds. However, it is unrealistic that there are no shared resources. Thus we added the following plausible resource constraints:

```
/* Resources
0  SensorReadings
1  GPS
2  Vision
3  Log
4  Drives and Steer
*/
const ResourceSet CLAIM[N] = {
(1<<0)+(0<<1)+(0<<2)+(0<<3)+(0<<4),  // FO Gyro
(1<<0)+(0<<1)+(0<<2)+(0<<3)+(0<<4),  // Magnetometer
(0<<0)+(1<<1)+(0<<2)+(0<<3)+(0<<4),  // GPS
(1<<0)+(0<<1)+(0<<2)+(0<<3)+(0<<4),  // Sonar
(0<<0)+(0<<1)+(1<<2)+(0<<3)+(0<<4),  // Vision
(0<<0)+(0<<1)+(0<<2)+(1<<3)+(0<<4),  // Log
(1<<0)+(1<<1)+(1<<2)+(1<<3)+(1<<4),  // Supervisor
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),  // Wheel drive 1
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),  // Wheel steer
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4),
(0<<0)+(0<<1)+(0<<2)+(0<<3)+(1<<4) ;
```

Here it takes about 15 minutes to generate a schedule. An attempt to increase the interference even more by increasing the run time for the Supervisor to 7 did not give any result after an hour.

The test case was also used, without the claims, in an experiment with the simple process model with a single region guarded by the readers-writers algorithm. Here it took about 30 minutes to find a feasible schedule.

6. IMPLEMENTATION

To verify that the proposed CMP cyclic executive results in a simple, easy to analyze system we have implemented the CMP cyclic executive on a Java processor.

6.1 System Prerequisites

One attractive property of a cyclic executive is that it does not need to deal with preemption and needs no interrupt from a timer. Only access to a passive clock is needed to release the individual tasks at the preplanned points in time. The more fine grain the clock resolution is the more flexibility is available for generation of the schedule. Former use of minor frames as basis for the static schedule where probably also motivated by the course grain granularity of the available clock (besides simplification of manually generated schedules).

A passive clock that ticks with the processor frequency is usually available in modern processors for embedded systems. On CMP architectures the individual clocks are driven by the same oscillator and they will not drift in relation to each other. We assume that frequency scaling is not used in a safety-critical system. On remaining issue is that the individual processor clock may not start at the same tie instant, due to the processor startup sequence. In that case a clock synchronization algorithm to measure the offsets of the individual ticks needs to be performed before mission start.

6.2 Implementation on JOP

We have implemented the proposed CMP cyclic executive scheduler in the context of SCJ on a CMP version of JOP [15, 12]. The scheduler is implemented in plain Java. All access to system level I/O devices are performed via hardware objects [16]. For the scheduling of the tasks only a passive, for all cores synchronous time base is needed. In the case of JOP, all cores have a local clock that ticks at 1 MHz for the scheduling decisions. The clocks start synchronous and are based on the same clock input.

If we want to achieve a tighter release jitter we can use the clock tick counter on JOP. For an extreme low jitter on the release (single cycle) a *deadline* instruction, which is available on JOP [17], can be used. A deadline instruction performs a busy wait in hardware until a programmed clock tick. With this hardware supported wait operations can be timed with processor clock resolution.

The CMP version of JOP boots with a single core that executes the main method. The other cores are running idle till enabled. The method that will be executed by the other cores is a Runnable that is set via a system function. Those Runnables contain the core local schedulers. At mission start the other cores are enabled and will start their schedules.

```
Runnable r1 = new Task(1, new RelativeTime(100, 0));
Runnable r2 = new Task(2, new RelativeTime(300, 0));
Runnable r3 = new Task(3, new RelativeTime(300, 0));

CyclicSchedule.Frame frame0[] = {
    new CyclicSchedule.Frame(
        new RelativeTime(50, 0), printer),
};

CyclicSchedule.Frame frame1[] = {
    new CyclicSchedule.Frame(
        new RelativeTime(100, 0), r1),
    new CyclicSchedule.Frame(
        new RelativeTime(300, 0), r3),
};

CyclicSchedule.Frame frame2[] = {
    new CyclicSchedule.Frame(
        new RelativeTime(300, 0), r2),
    new CyclicSchedule.Frame(
        new RelativeTime(100, 0), r1),
};

CyclicSchedule.Frame cmpSchedule[][] =
    {frame0, frame1, frame2};
CyclicSchedule sch = new CyclicSchedule(cmpSchedule);
```

Figure 3: Schedule definition of 4 tasks on 3 CPUs

The actual *scheduler* for the static cyclic executive is just a few tens lines of Java code and therefore too trivial to be described in the paper. However, the triviality of such a scheduler is a good argument for a cyclic executive for safety-critical systems. The complex part of generating the schedule is done offline, which will simplify the certification process.

The scheduler also contains detection of deadline overrun. On a overrun no task is interrupted, but the fact can be queried by the application.

In order to have a coherent view of the main memory, it must be ensured that the content of the main memory is updated. In standard Java, with properly synchronized code, this update may be performed on access to volatile variables and at synchronized blocks and methods. In our implementation we enforce the coherence by accessing a volatile variable before each task release. Therefore, the handling of synchronized methods and code blocks can be simplified in the JVM implementation.

6.3 A Simple Example

Figure 3 shows the usage of the cyclic executive framework. The tasks r1, r2, and r3 represent the example from Table 1, where one task needs to migrate between two CPUs. The tasks just add a *Hello* message to a output queue and simulate execution by a busy wait for their actual cost (100 and 300 ms). Additionally to this task set a printout task (printer) executes on core 0 and takes the messages from the queue and prints them to the console. In the current JOP CMP system only core 0, which also performs the system startup, is connected to I/O devices.

Figure 4 shows the output from the run of the example. Each tasks prints a message containing it's ID and on which processor it is actually running. From the output we can see that task t1 alternates between CPU 1 and CPU 2.

The example shows also a possible optimization to save space in the scheduling table. The individual cyclic schedules do not need to be all the same length. Just the longest schedule has to be a multiple of all other schedules. In the example the major period is

```
JOP start V 20091128
60 MHz, 1024 KB RAM, 1024 Byte on-chip RAM, 3 CPUs
A CPM cyclic executive scheduler example:
Hello from task t2 on CPU 2
Hello from task t1 on CPU 1
Hello from task t3 on CPU 1
Hello from task t1 on CPU 2
Hello from task t2 on CPU 2
Hello from task t1 on CPU 1
Hello from task t3 on CPU 1
Hello from task t1 on CPU 2
Hello from task t1 on CPU 1
Hello from task t2 on CPU 2
...
```

Figure 4: Console output of the example

400 ms, but the scheduler for CPU 0 contains only a single task that is executed every 50 ms. The longer schedule is a multiple of the CPU 0 schedule.

6.4 Departures From the SCJ Specification

The proposed cyclic executive API departs from the actual SCJ specification level 0 in following points:

CMP Chip-multiprocessors are not considered for level 0 of SCJ. Relaxing this restriction is the topic of the paper. To avoid multi-processing at SCJ level 0, the application just has to use a uniprocessor or use only one available processor in the static schedule.

Runnable In SCJ all tasks are represented as bound asynchronous event handlers with a period and a priority. In level 0 those two parameters are simply ignored as the schedule is given explicitly. In our implementation we used simple Runnables, a standard Java interface, for the application tasks. We argue that information that is not used represents dead code, which shall be avoided in certifiable applications. The Runnables of a level o application can easily be used at level 1 or 2 by invoking run() from a handler with the proper release parameters.

Task migration SCJ disallows task migration in level 1 to simplify the scheduling analysis and the scheduler itself. For a cyclic executive task migration can be handled easily and gives more freedom in the generation of the static schedule.

Frame data structure In SCJ a frame contains an array of handlers that are released in sequence within one time frame. In our definition of a Frame only a single Runnable can be defined. The argument for a set of handlers is to allow more flexibility in the task scheduling when it is known that tasks need different execution times in different application modes. However, this array of handlers just clutters the code to define the cyclic executive schedule. If this flexibility is needed, it can be easily implemented by defining a single handles that invokes the array of handlers that shall be executed in a single frame.

Overrun Even if WCET analysis shall be used to avoid any deadline misses the detection of overruns is usually part of cyclic executives. Therefore, we support the (late) detection of overruns. It can be queried from the application by a static method in the scheduler.

Current processor For testing it might be interesting that a task knows on which CPU it is running. Therefore, we have added a static method getCurrentProcessor() to the scheduler.

7. DISCUSSION AND CONCLUSION

We have presented an implementation of cyclic executives for chip-multiprocessors targeted for safety-critical applications where simplicity and predictability is of utmost importance. Schedules are generated using the facilities of the model checker Uppaal, where tasks are modelled by simple timed automata. Two extended models that handle resource constraints through static schedules are presented as well. Experiments with a realistic case of 16 tasks shows that the approach is feasible.

There is some indication that resource constraints makes it clearly more difficult to find schedules. Thus, for short critical sections, it may be more advantageous to use a spin-lock and increase the cost with a busy wait time corresponding to the cost of the critical section times the $M-1$, recall that M is the number of processors.

Further work include more systematic experiments to compare with uniprocessor preemptive scheduling, and preemptive scheduling on a multi-processor with static task partitioning and global scheduling. For that purpose, we intend to develop a parameterized version of the case study used in these experiments. Hopefully it can be reused as a benchmark for the kind of small to moderately large applications that are found in industry.

Acknowledgement

The comments of the reviewers have been very helpful in preparing the final version of the paper. This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

8. REFERENCES

[1] Ismail Assayad and Sergio Yovine. Modelling and exploration environment for application specific multiprocessor systems. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:433–434, 2007.

[2] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25, 1989.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical report, University of York, 2009.

[5] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[6] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[7] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[8] Herman Kopetz. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.

[9] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

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

[11] A. Metzner, M. Franzle, C. Herde, and I. Stierand. Scheduling distributed real-time systems by satisfiability checking. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 409 – 415, 17-19 2005.

[12] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted for publication, 2010.

[13] P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES99)*, pages 178–182, New York, May 3–5 1999. ACM Press.

[14] Klaus Schild and Jörg Würtz. Off-line scheduling of a real-time system. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 29–38, New York, NY, USA, 1998. ACM.

[15] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[16] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *Trans. on Embedded Computing Sys.*, accepted, 2010.

[17] Martin Schoeberl, Hiren D. Patel, and Edward A. Lee. Fun with a deadline instruction. Technical Report UCB/EECS-2009-149, EECS Department, University of California, Berkeley, October 2009.

[18] Andy J. Wellings. Multiprocessors and the real-time specification for java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.