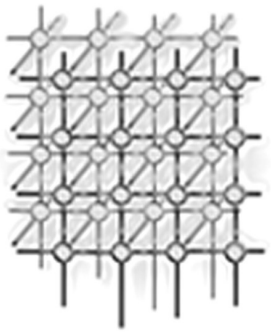
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Safety-Critical Java with Cyclic Executives on Chip-Multiprocessors

Anders P. Ravn[1,*,*] and Martin Schoeberl[2†]

[1] *Department of Computer Science, Aalborg University*
[2] *Department of Informatics and Mathematical Modeling, Technical University of Denmark*

SUMMARY

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. Mutual exclusion to access shared resources is guaranteed by including access constraints in the schedule generation. 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 schedule and avoid dynamic locking of shared variables. 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 e.g., [16] 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

*Correspondence to: Department of Computer Science, Aalborg University, Selma Lagerlöfsvej 300, DK-9220 Aalborg E.
*E-mail: apr@cs.aau.dk
†E-mail: masca@imm.dtu.dk
Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98–1846389

Received 3 December 2010
Revised 21 Februar 2011

- Easier predictable worst-case execution time (WCET), because preemption destroying cache content is avoided
- Simple context switch because it happens at predefined points of the program
- Fewer context switches because there are none caused by preemption
- Simple dispatcher

Disadvantages:

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

From a programmer's point of view, the constraints on periods are likely to be 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 other cores take care of tasks with a shorter period. Furthermore, the discipline of having minor frames defined by the greatest common divisor (GCD) of periods can be relaxed even on a uniprocessor. It was presumably introduced in earlier times 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. Locke [16] is essentially more negative towards cyclic executives than we are, and in particular, he points out the more painful software life-cycle that is caused when a static schedule has to be regenerated due to changes in the task set or task parameters. Here we see automated schedule generation through tools as a remedy.

Although dynamic scheduling offers greater flexibility, static scheduling with cyclic executives continues to exist. Even in the proposed safety-critical Java specification (SCJ) [10], 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 automatic generation of a static schedule for a cyclic execution model on a shared memory CMP. The contribution is an automatic generation of a static schedule that allows tasks to be scheduled at varying points of time, thus avoiding the straightjacket 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 pointed out in [16]; but it also introduces conflicts in access to shared data. We investigate solutions that involve static conflict resolution through constraints on the allowable schedules. The schemes include simple mutual exclusion but also readers-writers style exclusion which should eliminate some conflicts by having tasks execute a read, execute, and write sequence.

The schedules are generated with a model checker. Because the problem is known to be NP-complete, we may as well use tools that are built to tackle such problems. As demonstrated in the Times tool [1], a model checking engine is very useful for analyzing scheduling problems where there are additional constraints on the tasks, e.g. dependencies. Note that since we are generating schedules for the non-preemption case, our models stay within the decidable subset of timed automata. We do not

need to code a dynamic scheduler using bounded counting variables [7] or using over-approximations for stop-watches [5]. Neither do we need to consider communication costs as in [17].

A preliminary version of this research was presented in [21]. In this version, we have revised the task models for generating schedules thoroughly so that they reduce the state space used in exploring feasible schedules. We have added a systematic parameterized use case, which is synthesized from information derived from industry cases. Finally, we have updated background and related work.

In the following, we give a short background on related work and the work on safety-critical Java, and then we introduce the execution model in Section 3, followed by a description of schedule generation in Section 4. Section 5 gives some results for instructive examples of schedule generation, 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 [6]. 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 [3]. In the survey mentioned above, an algorithm by Horn [12] to build a schedule for a job shop problem is mentioned. It reformulates the problem as a linear programming problem that 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 [14] 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 [22]. Pop et al use a simulated annealing algorithm to find a schedule when an exhaustive search is too expensive [20].

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 KRONOS real-time model checker for such purposes in a similar way to ours, see for instance [2] for a recent result. Also Metzner et al solve the task allocation problem for distributed real-time systems with a SAT solver [18]. 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 are directly supported by the timed-automata model checker UppAal [15].

Finally we have the Giotto framework [11], which uses static scheduling within frames. However, the static schedule is used primarily to prevent I/O-jitter. In our approach we see the 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 [10]. Since safety-critical systems cover 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) [4]. The original RTS 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 [27]. 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 [16]. 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.[†]

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 [6] called homogeneous systems, because we consider M identical processors. The application consists of N periodic tasks τ_i. Each task has a period T_i, a deadline D_i, and a worst-case execution time (cost) C_i. The deadline is assumed to be less than or equal to the period. Each release k of a task can run on one of the M processor cores.

A schedule is fully defined by: 1) the start times s_{ik} of the releases of all tasks, with $k = 0, \ldots, SCM_{1 \leq i \leq N}(T_i)$, where SCM is a function that computes the smallest common multiple of its

[†]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.

arguments; and a processor assignment $a_{ik} \in P = \{1 \ldots M\}$ 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

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

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

Note that the processor assignment permits job-level migration. Compared to distributed or loosely coupled systems, task migration between cores on the same chip is cheap. And it offers some flexibility. Disallowing migration ($a_{ik} = a_{il}$ for all k, l) is an additional option that reduces the set of feasible schedules. 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 [11] 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 or for ensuring mutual exclusion on shared resources the same assumption is needed. For CMPs this synchronization comes for almost free, because the individual clocks can be driven from the same oscillator.

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 the 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, with and without the notion of individual resources
- Use non-blocking queues between individual tasks
- Implement a transaction model. That is far from simple, so we will not consider it further
- Implement locks (Java synchronized). The possible blocking time has to be included in the WCET of individual tasks
- Use of a global spin lock

When using locks, blocking time, due to cross-core locking, needs to be considered in the individual task's execution time. For a static schedule on each core, waiting on a lock does not introduce any task switching. Therefore, the tasks simply perform a spinning wait for the lock. To bound the blocking

time, the waiting tasks need to receive the lock in FIFO order. The maximum blocking time for a critical section s is bounded by the number of tasks n that share an object and the number of cores M by $max(n-1, M-1)t_s$, where t_s is the WCET of the critical section. For nested locks the transitive dependency needs to be taken into account.

Usage of a global spin lock is very simple to implement. However, it introduces artificial dependencies between tasks. The WCET of a task with critical sections is increased by maximal $M-1$ blocking times per critical section.

The two locking schemes make estimation of the WCET harder. Thus, in our implementation, we use constraints when generating the schedule. Resources are modelled by boolean values R_v, $v \in 0,\ldots,V$. When a task i is executing, it has a given set of used resources $CLAIM_i$ that 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

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

This simple scheme can be refined to distinguish between read and write access and allow multiple readers when no write is in progress. We have also included a schedule generation for the simple task model that essentially implements a readers-writers algorithm for access to the shared resources.

3.2. Implementation Internal Resources

Shared resources in libraries, the Java virtual machine (JVM), and the operating system (OS) have to be represented in the CLAIM vector as well. That is not different from considering locks in all software layers for worst-case response time analysis in preemptive scheduled applications. However, the cyclic executive with the restrictions of safety-critical Java simplify the runtime system greatly. Let us assume, for the discussion, that no dedicated OS is needed: the runtime system is part of the JVM.

Within a JVM there are three areas that usually need protection with locks: (1) dynamic class loading, (2) thread related functions, and (3) dynamic memory management (synchronization for object allocation and synchronization between write barriers in the mutator threads and a concurrent garbage collector). Dynamic class loading is avoided in SCJ and for the cyclic executive without preemption no threads and the related synchronization is needed. The memory management is also simplified in SCJ. RTSJ style scopes are used instead of garbage collection. Individual handlers (tasks) have private scopes and the backing store for the scopes is already preallocated at mission initialization. Therefore, no synchronization for scope creation, scope entering and exiting, or allocation of new objects in scopes is needed.

Synchronization for allocation of objects is only needed if objects are allocated during the mission phase in the shared mission memory or in immortal memory. However, this is highly discouraged; shared object shall be created during mission initialization. To avoid all synchronization within the JVM, objects creation in the mission phase has to be restricted to private scopes.

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

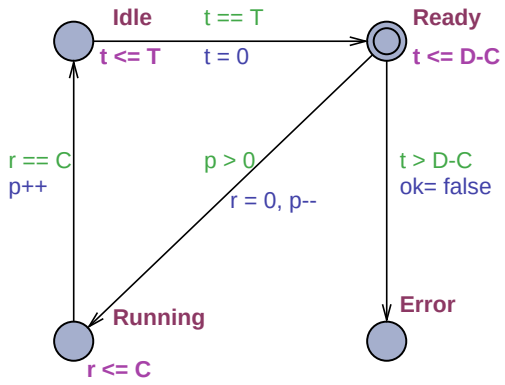


Figure 1. Simple model of a task

the application by an automaton τ_i, which is parameterized with its release parameters: the period T_i, the execution time C_i, and the deadline D_i.

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 Ready; in this simple model all tasks become ready at the start of the schedule. Start offsets are discussed later under the model with jitter constraints.

A task must transit from Ready to Running at the latest when the remaining time corresponds to the deadline minus the cost. This is enforced 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 on the transition to Idle ensure that this must happen.

If the schedule investigated is infeasible, the task will go from Ready to Error and reset the ok flag, if it is too late to enter Running. On the transition from Ready to Idle the number of free processors is increased. In Idle 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 task set. For this task set one should find a feasible schedule for a full hypercycle (hc = SCM(T_i)). In order to check when this time has passed, we introduce a global clock gt which is never reset.

There is no scheduler controlling the transitions; the automata 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 the global clock gt has advanced to the hypercycle hc and no task failed:

Table I. Schedule for the example task set

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

```
E<> ok and gt = hc
```

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 of a generated schedule we use the task set of three tasks that needs task migration between two processors to be schedulable. For this we set up the parameters. First the number of processors

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

and then the task set

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

const int N = 3; // Number of tasks
typedef int [0,N−1] Id;
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) = Process(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 IDs.

The model checker has several options that determine the strategy and the result. In general, for these experiments, we use a search strategy that is *randomized, depth first*. It will return a schedule as the one given in Table I. With this setup it is easy to experiment with the parameters and 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. Limitations of Cyclic Executives

Cyclic executives are simple and with modern tools the dispatch tables can be generated automatically; but are they not inferior compared to dynamic scheduling mechanisms? In order to answer this question one could ask for utilization bounds as known for fixed priority and earliest deadline first preemptive schedulers. Alas, it is easy to give an example of a task set which cannot be scheduled under a non-preemptive scheme and which has an arbitrary low utilization. Consider a task τ_0 with $C = 1$ and $T_0 = D_0$ and another task τ_1 with $C = 2T_0$ and $T_1 = D_1$. Clearly, when τ_1 is running, τ_0 will be released at least once so that it cannot run before its deadline. Since the periods can be chosen freely, e.g. $T_0 = 200$ and $T_1 = 80000$, the utilization can be arbitrarily low, for the example 1%! The argument generalizes easily for more processors, for instance, with two processors add a third task τ_2 with $C_2 = 2T_1$ and an arbitrary T_2. Thus a utilization test for multiprocessors is not really feasible, in contrast to what can be done for preemptive scheduling; see e.g., the recent results in [9]. Yet, it is clear that these task sets with low utilization are artificial; they go to the extreme in long running tasks. When periods are reasonably commensurable, cyclic executives can handle most task sets, although one would like to have a more rigorous definition of "reasonably commensurable", but we are not able to find this general characterization.

It is well known that the dispatch table becomes very long when periods are not aligned, for instance if they are prime numbers. It may be doubtful whether those tables can be generated. In order to check this, we have done some experiments. The first is with the following task set with 5 tasks:

```
const TaskSet TS = { // T D C
  { 5, 5, 2 }, { 7, 7, 2 }, { 11, 11, 1 }, { 13, 13, 1 }, { 15, 15, 1} };
```

which has an utilization of 90 % and a hyperperiod $hc = 15015$ which is the limit of UppAal's 16-bit integers. The dispatch table would then contain 8600 entries. In the experiment, the off-line version of UppAal, verifyxta searched about 70.000 states[‡] and generated a 6 MB trace, which the UppAal GUI clearly refuses to display. A scaling of the experiment to two processors requires duplicating the task set (the model checker is ignorant of the fact that it could just duplicate the previous schedule). This fails to give a result in a reasonable time (4 M states explored, when stopped). Eliminating the last task with a period of 15 reduces the hyperperiod to 5005 for the 8 tasks. Then the experiment gives a positive result after exploring 80.000 states and generates a trace file of 4.7 MB. Doubling the number of processors again to 4 and doubling the task set as well to 16 tasks give a result after exploring 830.000 states and generating a 17.6 MB trace. With 8 processors and 32 tasks repeating the set { 5, 5, 2 }, { 7, 7, 2 }, { 10, 10, 1 }, { 10, 10, 1 }, the checker terminates after exploring 5.6 MB states with the verdict that no trace is found. This is not too surprising, because the average utilization is 95 %.

The experiments indicate that practically useful results can be reached with the model checking approach even with challenging period selections and a realistic number of tasks (32) and cores (8). Based on these experiments, a practically analyzable configuration keeps $N \cdot M \cdot S < 50.000$, where the dispatch table length S is in the order of to the hypercycle divided by the shortest period.

[‡]We do not give execution times, because they depend much on the computer used, while number of states searched is invariant. However, for the curious reader, we add that the experiment took the time to put the kettle on, but not to make tea. That is on a simple laptop computer.

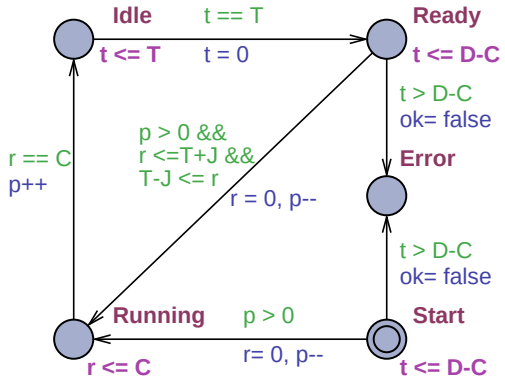


Figure 2. Task model with jitter constraint

If a task set cannot be scheduled, it may help to relax the deadlines, reduce the costs or increase the periods. It is clear that schedule generation is monotone in deadlines and costs. If a task set can be scheduled with shorter deadlines or higher costs, this schedule will do for the longer deadlines and smaller cost. It is not as easy to see that schedule generation is monotone with respect to the periods. Some anomalies could occur when periods are increased gradually. Some experiments have shown monotonicity, but a general argument is missing.

4.2. Jitter, Offsets and Delayed Start

Release jitter may be a concern with the generated schedule. 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. At the first release jitter is controlled by adding a start location Start, which corresponds to a Ready location without jitter constraint. This model is shown in Figure 2.

If desired, the transition from Start to Running could delay the first release and thereby all further releases with some relative offset DR by strengthening the guard with the condition t == DR. It is clear that an offset greater than D-C leads to Error.

An absolute offset with a delayed start requires an additional location before Start; this location would correspond to Idle and delay the start. However, when delayed starts are introduced, the query for a feasible schedule needs to be modified, so the trace ends after a full hypercycle following the maximal start delay. The best query we can find is E<> (ok and gt > MAXDELAY + 2*hc).

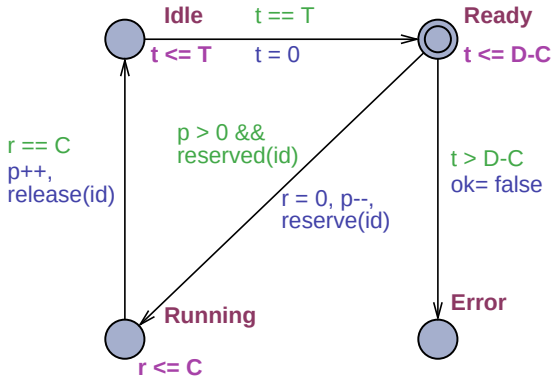


Figure 3. A task model with claims of shared resources.

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 set. 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 such modifications for future users and proceed to discuss implementation of constraints on access to shared resources.

4.4. Constraints on Shared Resources

In a very simple scheme, a global bit vector free represents shared resources and each task has its bit vector CLAIM.

```
/* configuration for shared resources */
const int RESOURCES = 5;
typedef int [0,1] ResourceSet[RESOURCES];
const ResourceSet ALL = {1, 1, 1, 1, 1};

/* free resources */
ResourceSet free = ALL;
```

```
/* claims for active tasks */
const ResourceSet CLAIM[N] = {
  {0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1},
  {1, 0, 1, 1, 0}  };
```

In the automaton, see Figure 3, the guard on the transition from Ready to Running is strengthened with a check reserved(id). It ensures that the resources are available, and the update reserve(id) claims them. When leaving for Idle, they are released by release(id).

Adding resource constraints reduces the possible parallelism of the task set. Especially when most of the execution time of a task is spent without accessing shared resources. To increase parallelism a task can be structured as a simple task [13]. A simple task has three phases: (1) read shared state, (2) execute, and (3) write shared 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 4.

The splitting of tasks into the read-execute-write subtasks is combined with the resource control with the resource vector CLAIM that is split into a READ_CLAIM and a WRITE_CLAIM. This increases the schedulability of the tasks compared to a single global resource.

5. Experiments with Application Style Cases

In order to see how 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},  {   50,   50,   2},     // Wheel 1 drive and steer
{   50,  50,   2},  {   50,   50,   2},     // Wheel 2
{   50,  50,   2},  {   50,   50,   2},     // Wheel 3
{   50,  50,   2},  {   50,   50,   2}      // Wheel 4
};
```

The utilization is 82 % and a schedule for it is found quickly. A variant is given by changing C = 8 to C = 17 for the GPS task. It is infeasible, when using the minor cycle approach, which was used in the original application. 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. In order to investigate what load can be accommodated, the cost C of the supervisor is increased. For

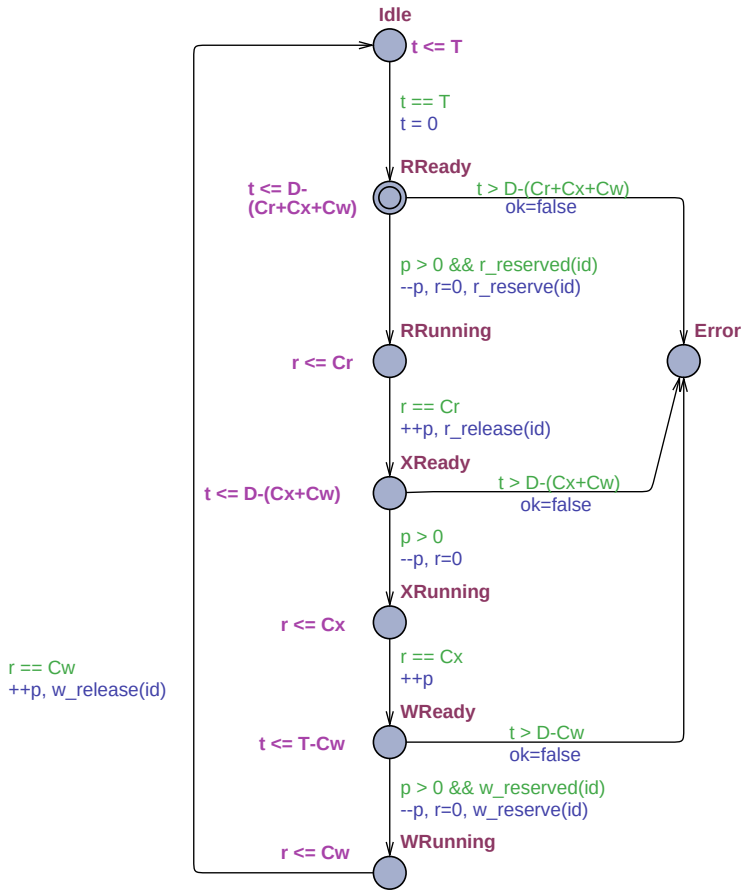


Figure 4. A task model with read, execute, and write phases

C = 6 (utilization 97 %) the task set is still feasible, but that is pure luck. A dual processor case is built by reducing the periods for the tasks above and increasing the cost values. In a further experiment such a dual core version with a utilization of 134 % is used.

However, it is unrealistic that there are no shared resources and these have to considered for the multicore case. Thus we added the following plausible resource objects:

```
/* Resources
   0  Sensor Readings
   1  GPS
   2  Vision
   3  Log
   4  Drives and Steer set-points */
```

with the following claims

```
const ResourceSet CLAIM[N] = {
  {1, 0, 0, 0, 0}, // FO Gyro
  {1, 0, 0, 0, 0}, // Magnetometer
  {0, 1, 0, 0, 0}, // GPS
  {1, 0, 0, 0, 0}, // Sonar
  {0, 0, 1, 0, 0}, // Vision
  {1, 0, 0, 0, 0}, // Operator Input
  {0, 0, 0, 1, 0}, // Log
  {1, 1, 1, 1, 1}, // Supervisor
  {0, 0, 0, 0, 1}, {0, 0, 0, 0, 1}, // Wheel 1
  {0, 0, 0, 0, 1}, {0, 0, 0, 0, 1}, // Wheel 2
  {0, 0, 0, 0, 1}, {0, 0, 0, 0, 1}, // Wheel 3
  {0, 0, 0, 0, 1}, {0, 0, 0, 0, 1}  // Wheel 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 readers-writers process model. Here it took about 30 minutes to find a feasible schedule for the latter example.

5.1. A Generic Parameterizable Use Case

One may wonder whether the case above is typical. However, considering industrial cases that we are familiar with, it is not too far off the mark. In order to do further experiments with more cores, we have come up with the following generic task set which is parameterized by a size n, and a smallest period T. The task set is assembled from:

1. 2 n short tasks with period T, deadline equal to the period, and cost 1. They are typical sensor readings and actuator settings. If release jitter is a concern for the readings, it may be checked as shown in Section 4.2.
2. n medium tasks with a period $2T$ and a proportional cost 2. They typically represent control loops including sporadic reactions to events.
3. $\lceil n/3 \rceil$ longer running tasks with period $10T$ and cost 15. They represent intelligent sensor readings, remote logging, watchdogs etc.

Table II. Schedulability tests with T = 5

n	Tasks	Utilization	M	Feasible	States explored
1	4	70 %	1	ok	200
2	7	140 %	2	ok	500
4	13	280 %	4	ok	2.000
8	26	560 %	7	ok	7.000
12	39	840 %	10	ok	20.000
16	52	1120 %	13	ok	113.000

Table III. Schedulability tests with T = 10

n	Tasks	Utilization	M	Feasible	States explored
1	4	20 %	1	ok	150
2	7	35 %	1	ok	450
4	13	70 %	1	ok	1.350
5	16	87 %	1	ok	2.000
10	32	175 %	2	ok	77.000
15	48	262 %	3	ok	50.000

This set up gives an utilization that is bounded below by $3.5n/T$. The number of processors is in the following experiments determined by an average utilization below 90 %, that is, $M = \lceil 4n/T \rceil$. For this use case we perform a search for the minimum number of processors M that are needed to execute the schedule depending on the problem size n. The results are shown in Table II and Table III.

In both cases, the exponential blow up is observed. However, it occurs so late that it should not hinder practical use. Due to the randomized search technique, the state space size should be taken with some caution. They vary with how "good" the first few choices are. This is for instance seen in the second table ($T = 10$) for the last case ($n = 15$) which finds a solution faster than for the case with $n = 10$.

5.2. A Case where Static Scheduling does not Apply

One use case from the automotive domain is an engine controller.[§] The tasks are organized as periodic tasks and tasks that are phase aligned with the engine. The periods are 1 ms to read sensors, 10 ms for several control laws, and 100 ms for other tasks. The periodic task sets are statically scheduled.

[§]Private communication with Guenter Gschwantner.

Therefore, communication between these tasks needs no additional locks. The period of the phase aligned tasks depends on the revolving speed of the motor. Such tasks are e. g. responsible for gas injection and ignition. Therefore, they usually have a very short deadline. The communication between the phase aligned tasks and the periodic task needs locks to protect critical sections or just use single words that can be read and written atomically.

For the phase aligned tasks, it seems most feasible to use dynamic scheduling with for instance fixed priority preemptive scheduling, deadline monotonic priority assignment to the tasks which must be treated as sporadic event triggered. In a multicore setting, a hybrid scheme with one processor allocated for these sporadic tasks would probably be a safe engineering solution.

6. Implementation

To validate that the proposed CMP cyclic executive results in a simple, easy to analyze system we have implemented the CMP cyclic executive on the Java processor JOP [23]. The cyclic executive also fits well with the available WCET analysis tool WCA [26], as it ignores (like most other WCET tools) any preemption costs and the effect on the cache state on preemption.

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-grained the clock resolution is, the more flexible is generation of the schedule. The use of minor frames as basis for the static schedule in former times was probably motivated partly by the coarse grained granularity of the available clock (besides simplification of manually generating the schedule).

A passive clock that ticks with the processor frequency is usually available in modern processors for embedded systems. On CMP architectures the same oscillator drives the individual clocks and they will not drift in relation to each other. We assume that frequency scaling is not used in a safety-critical system. One remaining issue is that the individual processor clocks may not start at the same time 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 [23, 19]. The scheduler is implemented in plain Java. All accesses to system level I/O devices are performed via hardware objects [24]. 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 synchronously 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 [25], 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.

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 [8]. 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 5 shows the usage of the cyclic executive framework. The tasks r1, r2, and r3 represent the example from Table I, where one task needs to migrate between two CPUs. The tasks just add a *Hello* message, which includes the information on which core the task is running on, to an 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 6 shows the output from the run of the example. Each tasks prints a message containing its 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 cycle is 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 this paper. To avoid multi-processing at SCJ level 0, when it is allowed, 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.

```
Runnable printer = new Runnable() {
    public void run() {
        printMsg();
    }
};

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 5. Schedule definition for 4 tasks on 3 CPUs

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 0 application can easily be used at level 1 or 2 by invoking run() from a handler with 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 handler that invokes the array of handlers that shall be executed in a single frame.

```
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 6. Console output of the example

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 with the model checker UppAal, where tasks are modelled by simple timed automata. Two models that handle resource constraints through static schedules are presented as well.

The emphasis in this extended version of the paper has been to explore thoroughly the potential of using model checking to generate static schedules for realistic applications. Therefore, we present solutions to incorporate static scheduling of access to shared objects into the models. In Section 4.4 we introduce a simple model, where each object is claimed for the full duration of an execution, and a refined model, where an execution is decomposed into a sequence of read, execute, and write phases, and the claims are refined into read and write claims. This is used in a readers-writers reservation algorithm for generating the static schedule.

There is some indication that resource constraints make it more difficult to find schedules. Thus, for short critical sections, it may be advantageous to use a spin-lock and increase the task's cost with a busy wait time corresponding to the cost of the critical section times the maximal number of potentially waiting processors ($M-1$) or maximum number of waiting tasks ($n-1$) that share a critical section.

Furthermore, we have investigated systematically the practical limits of a model checker in generating schedules. For that purpose, we have conducted experiments with task sets with relatively prime periods, that generate very long dispatch tables in Section 4.1. Yet, practically useful task sets can be handled. In Section 5, these investigations are continued with a realistic use case of 16 tasks and most systematically with a generic parameterized task set. Both sets of experiments indicate that model checker based schedule generation is practical.

7.1. Advantages and Disadvantages

In the introduction we listed advantages of cyclic executives as motivation for this research. As far as determinism, simple context switch, fewer context switches, and a simpler dispatcher are concerned, the implementation in Section 6 displays these properties. We further said that the absence of interrupts to implement preemption preserves cache content. This is implicitly demonstrated by the implementation. Since the whole application becomes a sequential program for each core, WCET estimation becomes feasible with static analysis or quantitative model checking with a precise model of caching. In this context it is worth noting that state-of-the-art WCET tools, like aiT from AbsInt, require exactly the same: "A task must be a sequentially executed piece of code".[¶]

Furthermore, we listed non-interference as an advantage, and this is really the same as saying no preemption, which may cause some disadvantages. The primary, theoretical disadvantage is the constraint on the maximal cost C that can be allowed. Here multiple cores turn out to be an advantage, because they relax the constraints as discussed in Section 4.1. It is also clear that careless selection of relatively prime periods will make schedule generation practically infeasible; but we seriously question whether there are practical cases where the periods cannot be chosen so they are multiples of a few base periods.

A more serious issue with the absence of preemption is that it will prevent implementation of a bandwidth server. Frame overruns remain an issue, but they should be prevented by checking the WCET, at least for critical applications. They would be problematic under other scheduling schemes as well. The more serious disadvantage is that cyclic executives are inflexible, so it may be impossible to adapt to sporadic tasks with short deadlines as illustrated by the case with varying periods in Section 5.

If the application needs such sporadic tasks or bandwidth servers, cyclic executives are not to be recommended. An engineering solution might be to allocate one core statically to such tasks and use a preemptive dynamic scheduler for that. This would be analogous to allocating I/O with interrupts to a dedicated I/O-processor core. When the cores have separate caches, the remaining cores can run cyclic executives undisturbed. That is, if communications between the partitions are handled by lock-free mechanisms or with very short spin-lock sections.

7.2. Further Work

Our conclusion is that cyclic executives may be used to advantage in safety-critical applications. There does not appear to be major obstacles to generating dispatch tables for realistic small to medium

[¶] `http://www.absint.com/ait/features.htm`, visited 2011-02-19

scale applications; but in order to get a solid safety case, the following tool developments would be beneficial:

1. A program that can extract release parameters from the handlers in a SCJ compliant application. It should be fairly straightforward to develop, if arguments to the parameter objects are restricted to constants.
2. A program that can extract the claim sets for handlers from their use of shared objects. This is a more demanding abstract interpretation analysis. In particular, it may be difficult to do it, if it has to be refined to read and write claims.
3. An adapter to the verifyta schedule generator such that dispatch tables are directly generated. Programs of that sort have been developed for other applications, so it should be doable.
4. A program that checks the dispatch tables using the original release parameters and the claims. Such a checker would remove the need to certify correctness of the model checker, which would be a major challenge.

This is what we foresee as our further directions. Furthermore, we believe that the generic application cases may be useful in further experiments and developments of hybrid approaches to multicore processor scheduling.

Acknowledgement

The comments of the reviewers have been very helpful in preparing the final version of the paper. Furthermore, we would like to thank Tobias Schoofs and Guenter Gschwantner for providing us with information on typical task sets in the aerospace and automotive domain, which guided the setup of the generic use case example. This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

REFERENCES

1. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 460–464, London, UK, 2002. Springer-Verlag.
2. 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.
3. Theodore P. Baker and Alan C. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25, 1989.
4. 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.
5. Alexandre David, Jacob Illum Rasmussen, Kim Guldstrand Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1, 2009.
6. Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *accepted for publication in ACM Computing Surveys*, 2011.
7. Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354:301–317, March 2006.
8. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.
9. Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:165–174, 2010.

10. 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.
11. 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.
12. W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
13. Herman Kopetz. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.
14. Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
15. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.
16. C. Douglas Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.
17. Jan Madsen, Michael Reichhardt Hansen, and Aske Wiid Brekling. A modelling and analysis framework for embedded systems, 2009.
18. 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.
19. Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.
20. 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.
21. Anders P. Ravn and Martin Schoeberl. Cyclic executive for safety-critical java on chip-multiprocessors. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 63–69, New York, NY, USA, 2010. ACM.
22. 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.
23. Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
24. Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, accepted, 2010.
25. 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.
26. Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.
27. 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.