An Open Framework for Real-Time Scheduling Simulation

Thorsten Kramp, Matthias Adrian, and Rainer Koster

Distributed Systems Group, Dept. of Computer Science
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{kramp,koster}@informatik.uni-kl.de

Abstract. Real-time systems seek to guarantee predictable run-time behaviour to ensure that tasks will meet their deadlines. Optimal scheduling decisions, however, easily impose unacceptable run-time costs for many but the most basic scheduling problems, specifically in the context of multiprocessors and distributed systems. Deriving suitable heuristics then usually requires extensive simulations to gain confidence in the chosen approach. In this paper we therefore present FORTISSIMO, an open framework that facilitates the development of taylor-made real-time scheduling simulators for multiprocessor systems.

1 Introduction

Real-time systems are defined as those systems in which correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Predictability is therefore of paramount concern with the scheduling algorithm being responsible for deciding which activity is allowed to execute at some instant of time so that the maximum number of tasks meet their deadlines. Unfortunately, optimal scheduling decisions easily become prohibitively expensive at run time or even computationally intractable, specifically for multiprocessors and distributed systems [15]. In these cases, heuristics may serve as viable alternatives, providing 'good enough' behaviour at acceptable run-time overhead. While certain properties of sophisticated heuristics can be derived analytically, it is often desirable to verify these results or even to find new approaches empirically. Thus, a customisable and extensible testbed is needed for observing the behaviour of a scheduling algorithm under well-controlled conditions. Such a scheduling simulator must provide enough infrastructure to let the real-time researcher concentrate on the details of the scheduling algorithm and yet must be open to new requirements. That is, in addition to a powerful dispatching core flexible load generators and statistics gathering facilities are needed.

By now, however, real-time scheduling simulators have been commonly build with a particular scheduling problem or execution environment in mind [11, 16]. In this paper we therefore present FORTISSIMO, an open object-oriented framework not exclusively aimed at simulating a particular class of scheduling algorithms but to serve as a starting point for the development of taylor-made real-time scheduling simulators for multiprocessor architectures [8]. Consequently,

FORTISSIMO is not a ready-to-run application, yet offers a frame of ideas to work in. Short of the concrete scheduling policy the framework consists of a number of ready-to-use components for workload creation, integration with dispatchers, and collecting run-time statistics. These components are realised as well-documented C++ classes and serve as the base from which the adaptation of FORTISSIMO to specific simulation requirements evolves. Thus, FORTISSIMO tries to support the real-time architect by coping with various scheduling paradigms rather than forcing him or her into a single notion.

Among the scheduling paradigms explicitly considered for hard real-time systems are static table-driven approaches such as cyclic executives [12], static priority-driven and dynamic best-effort policies such as rate monotonic scheduling or earliest deadline first [9], and dynamic planning-based strategies such as the SPRING scheduling-algorithm [14]. Task semantics, however, is not limited to hard real-time environments. Support for aperiodic and sporadic real-time activities [5], reasoning with value functions [6, 17] as well as requirements derived from techniques such as skip-over scheduling [7], imprecise computation [10], and task pair scheduling [4] have been included.

The remainder of this paper is organised as follows. Section 2 discusses related work that partially has influenced some of our design decisions. Then, in Section 3, the architecture of FORTISSIMO as well as the communication between the components are described. Section 4 finally summarises our experience and briefly outlines future work on FORTISSIMO.

2 Related Work

Naturally, concepts of other real-time scheduling simulation projects found their way into FORTISSIMO. Among the projects that have influenced our design, SPRING and STRESS come closest.

SPRING [14] is a research real-time operating system supporting multiprocessors and distributed systems. A project spin-off [3], the SPRING simulation testbed, has influenced the design of workload generation and scheduling components of FORTISSIMO. However, the primary focus of the SPRING simulator seemingly was to evaluate the planning-based dynamic-priority assignment policy used in SPRING. As a consequence, the simulator provides strong support for this kind of scheduling in a distributed environment, yet falls short when it comes to basically different scheduling strategies.

STRESS [1], in contrast, is a simulation environment for hard real-time systems consisting of a simulation core that is supplemented by a graphical front-end for control and display. The approach chosen comprises a full-featured simulation language to specify both the system environment and task semantics. The simulation engine is quite elaborate, including some feasibility tests and support for multiprocessing as well as networking; tasks may synchronise via critical sections or message-passing. Since STRESS is targeted at hard real-time systems there is no build-in support for soft-deadline or value-function scheduling and it is unclear whether the simulation language is rich enough to cope with imprecise

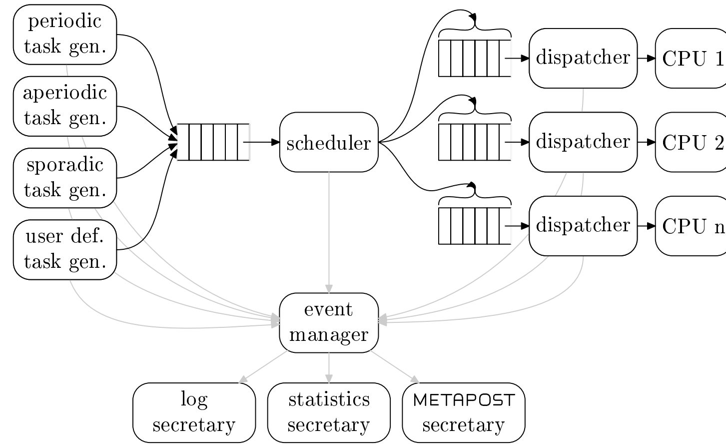

Fig. 1. Architecture of the framework

computing or task-pair scheduling, for instance. Task creation in FORTISSIMO, however, works in a similar way to STRESS.

3 Theory of Operation

As mentioned before, FORTISSIMO not only provides the basic infrastructure to build a real-time scheduling simulator suited for particular needs, but supports a number of scheduling paradigms right out of the box. Consequently, in order to add a new scheduler or task model, in most cases the real-time architect should need to refine or add only some specific classes rather than to redo everything from scratch. In FORTISSIMO, each class logically belongs to one of three independent modules, namely *workload generation*, *scheduling and dispatching*, and *gathering statistics*, with *tasks* and *events* serving as glue between these modules. The overall architecture is sketched in Fig. 1.

The workload component creates tasks according to user-defined patterns. Firstly, as part of the initialisation, the scheduler is allowed to check the feasibility of the specified task set as it will be generated from so-called *task generators*. Then, during simulation, the task generators create jobs for these tasks (e. g., instances of a periodic task) and place them into a global FIFO arrival queue. After removing a job from this queue, the scheduler can reject the job based on some feasibility test, accept and integrate it into its schedule, or react in a completely different way implemented by the user. An example would be putting the job aside and executing it only if additional execution time becomes available due to jobs that temporarily require less execution time than planned for.

As soon as a new job has been successfully scheduled, it is assigned to a *system dispatcher*, each dispatcher being exclusively responsible for one CPU. Again, scheduler and dispatchers communicate via queues with one ready queue per

dispatcher in which tasks are placed by the scheduler. The scheduler, however, retains full access to the ready queues to simply add or remove some task, or to perform complete reschedules if necessary. It is therefore the responsibility of the scheduler to sort the jobs in the ready queues to reflect its policy — the dispatchers simply execute the job that is currently at the front of their queue, automatically performing a context switch if a different task moves to the front at any time. Since CPUs are simply abstractions and time passes by as ticks from a logical clock, the execution of a task merely consists of decrementing an execution counter and updating internal busy/idle statistics.

When some job has completed execution, it is handed to the *statistics facilities*. Because some information of interest is often spread out over the complete lifetime of jobs and tasks, the statistics module also processes events from other components of the framework.

Based on this overview, the following sections give a closer look at each component. A longer version of this paper describes in more detail how schedulers can be implemented in FORTISSIMO and how the framework can be configured [8].

3.1 Taks Model and Workload Generation

Workload generation in FORTISSIMO is split among independent *task generators*, each one responsible for the generation of a single class of tasks. Readily available are generator classes for periodic tasks whose jobs re-arrive by some fixed amount of time, sporadic tasks whose frequency is limited by some minimum inter-arrival time, and aperiodic tasks whose arrival pattern is modelled by some stochastic assumptions. In addition, a user can create completely new task generators or customise the available ones via inheritance to produce workload patterns currently not explicitly supported.

Timing parameters of a task include its average-case computation time, its worst-case computation time, and its deadline; the first invocation of a task may be delayed by some initial offset to construct arbitrary task phasings, in order to prevent or enforce critical instants, for example. Furthermore, a directed precedence graph without cycles may be used to explicitely define predecessor/successor relationships. The basic classes of hard, firm, and soft constraints are employed categorising a deadline miss as resulting in a catastrophy, in the computation being useless, or in a degraded quality of service, respectively. Whenever this scheme is insufficient, two value functions per task may be used to describe the value of finishing the task up to and after its deadline. Each task may be assigned a base priority during setup while at run time an additional temporary priority per task can be used to support dual-priority scheduling [2] and priority-inheritance protocols [13], for instance. Besides these fundamental paradigms, skip-over scheduling, the notion of imprecise computations, and task-pair scheduling are also readily supported. While in FORTISSIMO skip-over scheduling is limited to periodic tasks, support for imprecise computations and task-pair scheduling is available for both periodic and sporadic tasks.

To assess the behaviour of scheduling algorithms many simulation runs with varying load patterns are needed. Hence, virtually all task characteristics may

be chosen randomly by FORTISSIMO according to given stochastic distributions. Parameters such as arrival patterns and actual computation time may vary for each job. Additionally, for a sequence of simulation runs changing task set characteristics may be specified.

3.2 Scheduling and Dispatching

Scheduling Algorithms are not built into FORTISSIMO, but have to be implemented and linked by the user. Schedulers, however, can be derived from a base class `Schedule` providing some default behaviour that can be customised selectively. We believe that this approach, besides promising some additional flexibility, allows analysing the computation time of the scheduler itself, already at the simulation stage.

A typical scheduler might work as follows within FORTISSIMO. The scheduler is invoked every tick of the logical clock and, provided it implements a preemptive algorithm, may perform a reschedule in response. If no new jobs have become ready since the last tick, the scheduler then falls asleep again until its next invocation. If new jobs have arrived, it removes these jobs one by one from the global arrival queue. For some algorithms providing guarantees, then a run-time admission test is performed. If the new job cannot be executed without jeopardizing the deadlines of either the new job itself or already guaranteed tasks, it is rejected and usually removed from the system. Otherwise, a new schedule must be constructed comprising the jobs already scheduled as well as the new job. For this, the scheduler typically has to retrieve the jobs already accepted and scheduled from the dispatchers' ready queues. Then, the jobs are sorted and re-inserted into the individual ready queues, possibly causing context switches.

Like the scheduler, dispatchers are invoked every tick of the logical clock. At any time, the dispatcher will run the job that is currently at the front of its ready queue, which subsequently becomes the active job until it terminates normally, the scheduler aborts the job for some reason, or the dispatcher's ready queue has changed.

Finally, jobs are run on *virtual processors*. Execution is simulated simply by decrementing the remaining execution time of the running job. In future versions, a more powerful processing model may, for instance, take interrupts and context-switching overhead into account.

3.3 Logging and Statistics

Whenever an important action is executed within the framework, this is signalled by an event. Each event carries the relevant information about the time and cause that lead to its creation, supplemented by additional data as needed. An *event manager* uniformly collects and distributes these events to so-called *secretaries*, which are registered with the event manager for certain types of events. Various types of action can be taken by a secretary upon arrival of a new event. Simple log secretaries just write a formatted line onto some output device, other secretaries

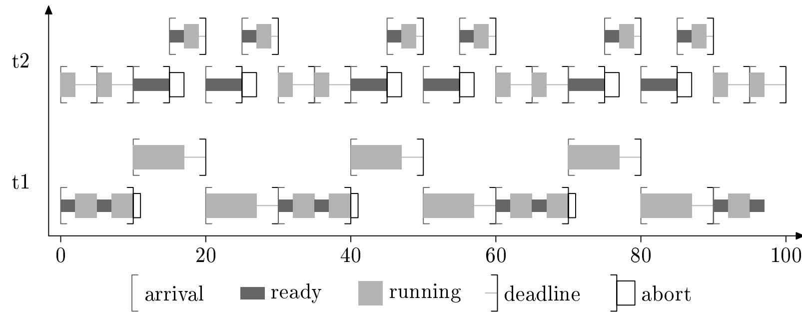

Fig. 2. Skip-over scheduling simulation run

may update some kind of statistical analysis data, and even more sophisticated ones may act as a gateway transforming the event into messages for a graphical user display. At the time of writing, secretaries for logging events, for collecting statistical data, and for visualizing a simulation run as a MetaPost figure are implemented.

Fig. 2 shows an example run of a skip-over scheduler that tolerates missed deadlines to a certain degree provided 'most' of a tasks deadlines are met [7]; a skip parameter s per task denotes the tolerance of that task to missing deadlines such that at least $s-1$ task instances must meet their deadlines after missing a deadline. The skip parameter of tasks t_1 and t_2 is set to 3 and 2, respectively; that is, after one aborted job of t_1, two jobs of t_1 must be executed in time, and no two successive jobs of t_2 may be aborted.

4 Conclusions

In this paper we have presented Fortissimo, an open object-oriented framework to simulate the scheduling of real-time tasks. The versatility of Fortissimo has been verified by implementing a wide range of fundamentally different scheduling policies such as rate-monotonic scheduling, earliest deadline first, the sporadic server algorithm, an imprecise computation policy, skip-over scheduling, and task-pair scheduling. Although the task model already provides a sound basis, we intend to add support for critical sections, resource reservation, task semantics including inter-task communication, and more elaborate precedence relations to the scheduling core. Furthermore, in addition to the multiprocessor support already implemented, an infrastructure to simulate real-time scheduling in distributed systems is under development. A graphical user interface, finally, will increase the ease the use of Fortissimo and illustrate behaviour of scheduling policies at run time; for the latter, the event mechanism already provides the necessary internal hooks. Despite these loose ends, however, we believe that even the scheduling core as described in this paper might already serve real-time architects to develop taylor-made simulators based on Fortissimo to evaluate their algorithms and heuristics.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. STRESS: A simulator for hard real-time systems. *Software Practice and Experience*, July 1994.

[2] R. Davis and A. Wellings. Dual-priority scheduling. In *Proceedings of the Sixteenth Real-Time Systems Symposium*, pages 100–109, 1995.

[3] E. Gene. Real-time systems: Spring simulators documentation, 1990. http://www-ccs.cs.umass.edu/spring/internal/spring_sim_docs.html.

[4] M. Gergeleit and H. Streich. Task-pair scheduling with optimistic case execution times—An example for an adaptive real-time system. In *Proceedings of the Second Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, February 1996.

[5] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time Systems*, 7(9):31–67, 1995.

[6] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the Sixth IEEE Real-Time Systems Symposium*, December 1985.

[7] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of Sixteenth IEEE Real-Time Systems Symposium*. IEEE, 1995.

[8] T. Kramp, M. Adrian, and R. Koster. An open framework for real-time scheduling simulation. SFB 501 Report 01/00, Department of Computer Science, University of Kaiserslautern, Germany, January 2000.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[10] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[11] J. W. S. Liu, J. L. Redondo, Z. Deng, T. S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. K. Shih. PERTS: A prototyping environment for real-time systems. In *Proceedings of the Fourteenth Real-Time Systems Symposium*, pages 184–188. IEEE, December 1993.

[12] C. D. Locke. Software architectures for hard real-time applications: Cyclic executives vs. fixed-priority executives. *Journal of Real-Time Systems*, 4(1):37–53, 1992.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. Technical Report CMU-CS-87-181, Computer Science Department, Carnegie Mellon University, 1987.

[14] J. A. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for hard real-time operating systems. *IEEE Software*, 8(3):62–72, May 1991.

[15] J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[16] A. D. Stoyenko. A schedulability analyzer for Real-time Euclid. In *Proceedings of the Eighth Real-Time Systems Symposium*, pages 218–227. IEEE, December 1987.

[17] H. Tokuda, J. W. Wendorf, and H.-Y. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of the Eighth IEEE Real-Time Systems Symposium*, December 1987.