CYCLIC DISPATCHING WITH PERIODIC OR END-TO-END TIMING CONSTRAINTS

Axel W. Krings, University of Idaho, Moscow, ID
and
M. H. Azadmanesh, University of Nebraska at Omaha, Omaha, NE

Abstract

This paper considers non-preemptive scheduling and dispatching in distributed real-time applications where the workloads are executed repeatedly. Specifically, we consider periodic and end-to-end scheduling. In *periodic* scheduling workloads must execute once during each repeating fixed duration time slot. In contrast, in *end-to-end* scheduling the timing constraints of repeating instantiations of the workload are relative. We present a dispatching environment for these applications that takes advantage of two main properties of list scheduling, namely simplicity and low run-time overhead. However, an inherent problem in list scheduling are so-called timing anomalies, where early finishing tasks may cause other tasks to miss deadlines. By modifying workloads using special tasks called phantom tasks, one can derive task systems that are capable of implementing periodic or end-to-end behavior while taking advantage of a variety of adaptable existing safe algorithmic dispatching solutions with different run-time complexity.

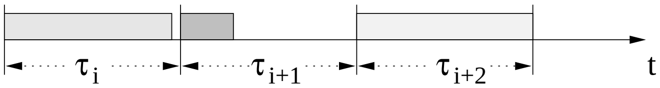
Introduction

Periodic tasks, i.e. tasks that execute or are repeated periodically on computers, have been the subject of much research. Famous solutions in preemptive environments have been derived that are built on rate-monotonic solutions [7]. However, for applications distributed over networks non-preemptive solutions might be desirable. Furthermore, whereas periodic scheduling is normally considered in task systems with independent tasks, we are interested in extending the concept of "periodic" to systems of tasks. These systems or workloads can be represented by task graphs in which each node is a task to be computed on some computing node and edges represent precedence constraints.

When dealing with the problem of repeatedly executing tasks or task systems two flavors of repetitions arise, i.e., Periodic Scheduling and End-to-End Scheduling.

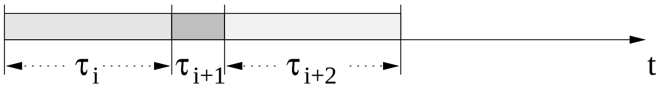
Periodic Scheduling: A task or workload is repeatedly executed during fixed time-slots. The time from one instantiation of the workload to the next is called the *period* denoted by $\mathcal{T}$. Periods are usually fixed, e.g., a control function may need to be executed "periodically" within each 1 ms time-slot. If task durations are run-time dependent, one inherent property of periodic scheduling is that results of two consecutive instantiations of the workload may be further than the period $\mathcal{T}$ apart. Such situation can be seen in **Figure 1** where workloads are represented by shaded boxes. Note that $\mathcal{T} = \mathcal{T}_i = \mathcal{T}_{i+1} = \mathcal{T}_{i+2}$. The workload in $\mathcal{T}_{i+1}$ finishes early and the same workload in $\mathcal{T}_{i+2}$ uses its maximal time. As a result the finishing times in $\mathcal{T}_{i+1}$ and $\mathcal{T}_{i+2}$ are further apart than $\mathcal{T}$. It is easy to see that in the extreme case this could lead to differences in finishing times of almost $2\mathcal{T}$.

Figure 1: Periodic Scheduling



End-to-End Scheduling: Whenever the timing dependencies are based on the difference in allowable finishing times, one speaks of end-to-end scheduling. Thus, by its very definition, two consecutive finishing times are allowed to differer by at most $\mathcal{T}$. An example is a distributed control application in which actuator coordinates need to be updated no more than $\mathcal{T}$ time units apart. As can be seen in **Figure 2**, time intervals are not fixed anymore, e.g. $\mathcal{T}_i > \mathcal{T}_{i+1}$, however, all time intervals are implicitly bounded by $\mathcal{T}$.

Figure 2: End-to-End Scheduling



The Task System

We adopt the notation from [5], where the basic units of computation are tasks. Tasks consist of sequentially executing code. They may be run on any of M homogeneous computers, and may not be preempted. Each task T_i has an associated minimum and maximum computation time c_i^{min} and c_i^{max}, release time r_i at which the task becomes *ready* for execution, starting time s_i, finishing time f_i, and deadline d_i. Task dependencies are defined by a partial order, represented by a directed acyclic precedence graph.

Special tasks called *phantom tasks* have been used to model events external to a processor, such as delayed task release, non-transparent overhead or task synchronization [1, 4]. These tasks are fully incorporated into the precedence graph. Although they consume time, unlike *real tasks*, i.e., tasks in the regular sense, they consume no resources. As a result, phantom tasks may *always* be started upon becoming released. One can imagine phantom tasks to be executing on an infinite number of "phantom processors". Phantom tasks will be the mechanism to implement periodic and end-to-end dispatching.

The work presented here is based on the list scheduling paradigm. In *priority list scheduling*, whenever a processor becomes available, the run-time *dispatcher* scans the task list from left to right, and the first unexecuted ready task encountered in the scan is assigned to the processor. The dispatcher is distinct from the scheduling algorithm. The scheduler is executed only once at design time for the workload. At run-time, the dispatcher simply arbitrates tasks. The scheduling process supplies a priority list $\mathcal{L} = \{T_1, T_2, ..., T_n\}$.

In the case of a cyclic dispatcher, which is essential for periodic or end-to-end scheduling, the priority list $\mathcal{L}$ must be seen as a cyclic list as well. Therefore, let $\mathcal{L}_i$ denote the priority list in the i^{th} iteration, i.e. $\mathcal{L}_i$ is the list used in $\mathcal{T}_i$. This should not be confused with having a separate list $\mathcal{L}_i$ for each iteration i. There is only one "physical" list, i.e., the iterations can be implemented using simple flags or counters.

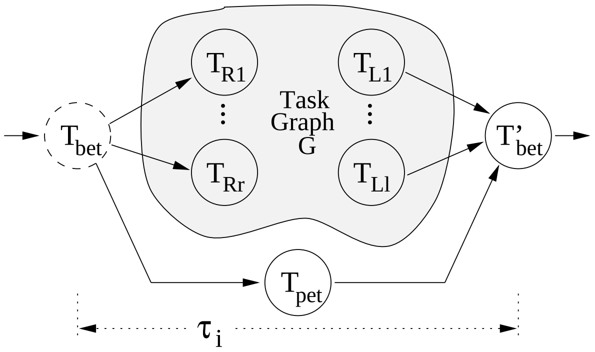
Objective

We are interested in low overhead dispatching solutions to cyclic workloads. Priority list scheduling has been of interest to many researchers due to its simplicity and low overhead. However, list scheduling in site of varying task durations can cause tasking instability, where early finishing tasks cause priority inversions that can result in missed deadlines [2]. Therefore, in order to address the hard real-time requirement of the application we need dispatching algorithms that guarantee avoiding instabilities. Many such algorithms with different assumptions and run-time complexity exist [1, 3, 4, 5, 6, 8, 9, 10]. However, in order to take advantage of them one has to first transform the workload to derive an appropriate task system and secondly adapt the dispatching algorithms to these new task systems. It should be noted out that the solutions presented in [1, 3, 4, 5, 6] already consider the phantom task concept.

Workload Transformation

Assume that the workload to be executed is represented by precedence graph $G = (V, E)$, where V is the set of tasks and E the set of edges. Furthermore, as shown in **Figure 3** define the root tasks in G, i.e., tasks with no predecessors in G, as the set $T^R = \{T_{R1}, ..., T_{Rr}\} \cap V$, and the leaf tasks in G, i.e., tasks with no descendents in G, as the set $T^L = \{T_{L1}, ..., T_{Ll}\} \cap V$. Now

Figure 3: Task Graph Transformation

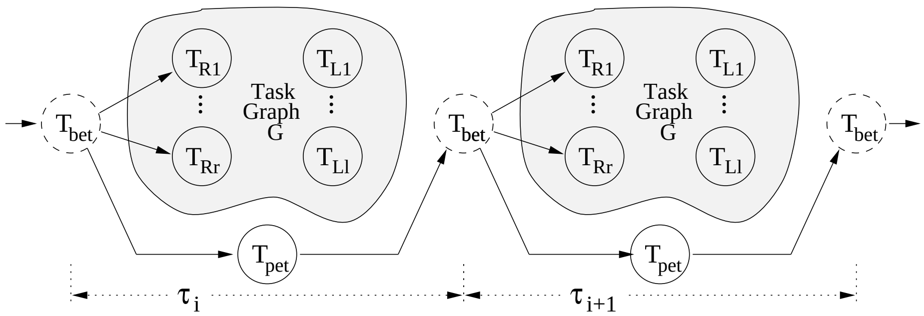


construct a graph $G' = (V', E')$ from $G = (V, E)$ by introducing two new mechanisms. Firstly, special phantom tasks called *boundary enforcement tasks*, T_{bet}, are defined. Tasks T_{bet} and T'_{bet} will be used to enforce the boundaries between two iterations of workload G. Secondly, a special phantom task called *period enforcement task*, denoted by T_{pet}, will be used to enforce periods. Not all tasks shown in **Figure 3** will be necessary, as will be described below.

Periodic Dispatching

First we discuss periodic scheduling, or more accurately periodic dispatching. **Figure 4** shows the principle. Two iterations are shown. Assume that task T_{pet} is initialized with computation time $c_{pet} = \mathcal{T}$. No task in $\mathcal{T}_{i+1}$ can start execution until task T_{bet}, which we initialize with $c_{bet} = \epsilon$, $\epsilon \to 0$, has terminated. However, T_{bet} in turn depends on T_{pet}. As a result periods are enforced. Task T_{bet} is drawn in a dashed circle to indicate that it could be collapsed into task T_{pet}.

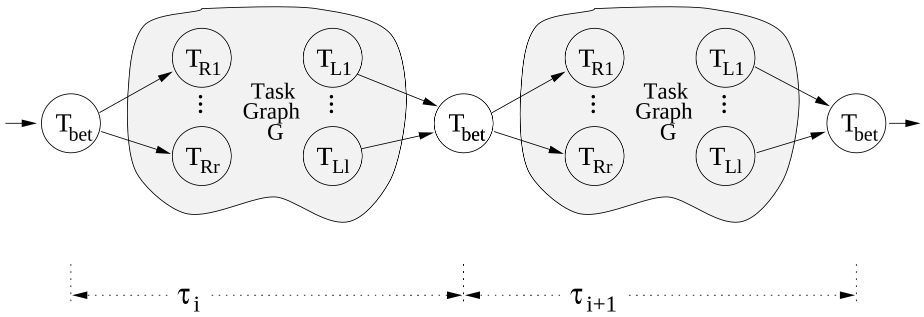
Figure 4: Periodic Dispatching



End-to-End Dispatching

In end-to-end dispatching there is no need to enforce periods as can be seen in **Figure 5**. Only boundary enforcement tasks are defined. Task T_{bet} is defined with $c_{bet} = \epsilon$.

Figure 5: End-to-End Dispatching



Dispatching Algorithms

The graph transformation of the previous two subsections simply puts the workload into a format that allows the adaptation of stable dispatching algorithms known from non-cyclic scheduling, e.g., those found in [3, 4, 6]. Here *stable* algorithm refers to dispatchers that cannot induce scheduling instabilities found in List Scheduling. Non-cyclic algorithms can be adapted to deal with the transformed workload representing periodic and end-to-end scheduling. The adaptation will be shown for stable dispatching algorithms presented in [3], but the same logic can be applied to modify the dispatchers in [1, 4, 5, 6].

When a processor finishes its current task, the traditional priority list dispatcher may scan the entire task list to find a ready task. Stability can be enforced at run-time by restricting the depth of the scan. The subset of *unstarted* real tasks scanned by the dispatcher are said to be in the *Scan Window*. The first task in the scan window is always the first unstarted real task T_u in the priority list. The number of unstarted real tasks scanned is called the *Scan Depth* D. In [3] two families (*basic* and *augmented*) of stable dispatchers are shown with general format

$$D = x - u + y. \tag{1}$$

Here x is the index of a task of special interest and u is the index of the first unstarted real task T_u. Furthermore, y defines the dispatching family with $y = 1$ for basic dispatchers and $y = I$ for the augmented dispatchers, where I is the number of idle processors at the time of the scan. The augmented dispatchers take advantage of extending the scan window by the number of idle processors, thus increasing the chances of finding a task ready to start. For each family of dispatchers four different definitions for x are presented each having different run-time complexity. Using a circular priority list $\mathcal{L}$ the algorithms can be easily adapted to consider iterations $\mathcal{L}_i$ by defining the scan-depth as

$$D = \begin{cases} x - u + y & \text{if } x \geq u \\ (x + N) - u + y & \text{otherwise} \end{cases} \tag{2}$$

where N is the number of real tasks in $\mathcal{L}$. Thus, the modifications to existing stable dispatchers simply address the multiple instance issue that is the result of "unrolling" the schedule into a cyclic schedule. With respect to the transformations shown in **Figures 4** and **5** phantom tasks T_{pet} and T_{bet} present limitations for the second case of equation 2, i.e., tasks T_{bet} inhibit $x < u$. This creates limitations for resource reclaiming as will be addressed next.

Resource Reclaiming

Resource reclaiming, often also called slack-time reclaiming, addresses using idle processor resources in order to run lower priority tasks. One of the most interesting features of the two transformed workloads described above is their lacking ability for resource reclaiming beyond the current iteration. The reason for this lays in the definition of T_{bet} as a predecessor to all root tasks T^R.

Excluding T_{bet} and its effect on precedence when addressing slack-time reclaiming may result in loss of synchronization. Assume that resources are idle waiting for tasks of the current iteration to be released, while independent tasks from successive iteration are available. By definition these available tasks must be in T^R. In the absence of T_{bet} the dispatcher will use independent tasks from higher and higher iterations in order to keep the processors occupied. However, it is very likely that only results that are computed close to the iteration in which they are used will be meaningful, e.g., it makes little sense to compute certain values way in advance and use these possibly stale values in future computations. As a result eliminating T_{bet} during slack-time reclaiming is not a general option.

An alternative is to associate T_{bet} with the iterations, i.e., defining T_{bet} for individual $\mathcal{L}_i$. Let i be the current iteration of the cyclic schedule. Then one can tune the ability of the dispatcher to look up to list $\mathcal{L}_{i+k}$, for a pre-determined value of k. Thus, for inter-iteration slack-time reclaiming T_{bet} is excluded from the next k iterations. As a result, k puts an upper bound on the maximum loss of synchronization.

The same approach, i.e., allowing the dispatcher to consider k iterations, can be used to let the dispatcher take advantage of the second part of equation 2, thus allowing general inter-iteration dispatching.

Conclusions

This paper presented simple graph transformations that allow using priority list scheduling and existing stable dispatching algorithms in real-time multiprocessor environments employing periodic or end-to-end scheduling. The adaptation of dispatching algorithms that avoid the problem of timing anomalies associated with priority list scheduling has been demonstrated using scan-window based stable dispatchers. Furthermore, a simple run-time mechanism has been introduced that allows inter-iteration slack-time reclaiming while defining an upper bound on the possible loss of synchronization. This mechanism can be used to allow inter-iteration dispatching.

References

[1] Deogun, J.S., R.M. Kieckhafer, and A.W. Krings, "Stability and Performance of List Scheduling With External Process Delays, " *Real-Time Systems*, Vol. 15, No. 1, July 1998, pp. 5-39.

[2] Graham R.L., "Bounds on Multiprocessor Timing Anomalies", *SIAM J. Appl. Math.*, Vol. 17, No. 2, Mar 1969, pp. 416-429.

[3] Krings, A.W., R. Kieckhafer, and J. Deogun, "Inherently Stable Real-Time Priority List Dispatchers", *IEEE Parallel & Distributed Technology*, Winter 1994, pp. 49-59.

[4] Krings, A.W., and M.H. Azadmanesh, *Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads*, Proc. HICSS-30, Vol I, 1997, pp. 616-625.

[5] Krings, A.W., and M.H. Azadmanesh, "Feasibility Tests for Stable Dynamic Scheduling", *Proc.* 10^{th} *International Conference on Parallel and Distributed Computing and Systems*, October 28-31, Las Vegas, Nevada, 1998, pp. 25-31.

[6] Krings, A.W., and M.H. Azadmanesh, "Avoiding Run-Time Infeasibility in Systems Containing Coupled Tasks", to appear in *INFOR (Information Systems & Operational Research).*

[7] Liu, C.L., and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Vol. 20, No. 1, Jan 1973, pp. 46-61.

[8] Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *JACM*, Vol. 14, No. 3, July 1967, pp. 439-465.

[9] McElvaney, M., et. al., "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *J. of Real-Time Systems*, (3), 3, Sep. 1991, pp. 275-305.

[10] Shen, C., et al, "Resource Reclaiming in Real-Time", *Proc. Sixth Real-Time Systems Symposium*, December 1990, pp. 41-50.