Scheduling Slack in MetaH

Abstract

A real-time implementation for allocating slack to aperiodic processes in MetaH [4] is nearing completion. The slack scheduling algorithm used is based on the slack stealer originally proposed in [2] with practical extensions to allow for support of process criticalities, multiple process streams (of different criticalities) competing for pooled slack and inclusion of run-time overheads in the slack functions. Areas in need of future work are also identified.

Introduction

A real-time implementation using a MetaH executive [4] of an algorithm for slack stealing [2], [3] is nearing completion. MetaH is a software architecture specification language that allows for specification of multiple processors, multiple modes, process criticalities, preperiod deadlines, among other things. The MetaH translator will automatically configure a real-time executive from a MetaH specification. The MetaH timing analyzer will also automatically generate and solve schedulability analyses, including run-time overheads for MetaH specifications. MetaH executives use Rate Monotonic Scheduling (RMS) for scheduling periodic processes on a processor.

The slack stealer initially proposed in [2] is an algorithm for “stealing” slack in a periodic process timeline in an attempt to provide aperiodic processes with more rapid response times. It is applicable to the general class of fixed-priority preemptive scheduling disciplines, which includes RMS. Due to the size restrictions of this submission, only a brief description of how the slack stealer works is given here. The reader is referred to [2] and [3] for a detailed definition. Intuitively, the slack stealer postpones the execution of periodic processes as long as possible without causing any of them to miss their deadlines so that an aperiodic process can execute earlier and hopefully achieve a shorter response time. More precisely, the amount of slack available from the start of a hyperperiod to the deadline of each task dispatched throughout the hyperperiod is first computed off-line and stored in a table that is used at run-time. The following three step procedure for the run-time calculation of available slack roughly characterizes the algorithm used in our implementation of slack scheduling in MetaH. First, a table look-up gives the amount of slack time available from the start of the current hyperperiod to the completion of the next periodic process (at a certain priority level). Next, this slack value is adjusted by subtracting off all aperiodic processing time and all idle time (of the appropriate priority level) that has occurred between the beginning of the hyperperiod and the time of the request. Last, the minimum of all slack values computed with equal or lower priority is taken to ensure that no lower priority periodic process will miss its deadline. In order to implement slack scheduling in MetaH, it was necessary to consider several practical problems in some detail, some of which we sketch in subsequent paragraphs.

Supporting Process Criticalities

In MetaH, both periodic and aperiodic processes can be assigned a static process criticality which indicates the level of importance the system designer attributes to the process. In MetaH, assigning a process \( p \) a criticality \( j \) means that all timing faults in processes with criticality less than \( j \) (criticalities are
numerically increasing) will not affect the amount of time allocated to $\rho$. In other words, timing faults in processes of lower criticality will not impact processes with higher criticality. During periods of transient overload, the criticality assignment is used to determine which tasks receive processor priority.

To implement support for process criticalities, the periodic process task set is first period transformed so the ranking of the transformed rate corresponds to the ranking of the process criticality. Here is a simple example to illustrate. Let the periodic task set be $T_1$, $T_2$, and $T_3$ with periods $T_1 = 102$, $T_2 = 202$, and $T_3 = 300$, and criticalities 1, 2, and 4 respectively. The transformed task set has $T'_1 = 100$, $T'_2 = 101$ and $T'_3 = 102$. Next the aperiodic processes are assigned a “slack-level” equal to the periodic process priority with the highest criticality not greater than the criticality of the aperiodic process. Continuing with our example, let $\alpha$ be an aperiodic process stream with criticality 3. Then $T_1$, $T_2$, and $T_3$ are assigned slack-level (or equivalently periodic process priority) 3, 2, and 1, respectively. $\alpha$ is assigned slack-level 2 since $T_2$ has priority 2 and it is the periodic process with the highest criticality less than that of $\alpha$. Priorities can be assigned within a slack-level when there are multiple aperiodic streams whose criticalities map to the same slack-level.

To support process criticalities (among both periodic and aperiodic processes), process timeouts must be enforced at run-time. A periodic process is allowed to overrun its deadline, but only by repeatedly requesting execution timeslices. For an untransformed periodic process, this may result in a missed dispatched (which the application may tolerate). Alternatively, the missed deadline can be treated as a failure where the process is restarted. For a transformed periodic process, only the last timeslice can result in determination of an incomplete process execution. Again, subsequent timeslices can be requested or a failure to miss a deadline might be noted. Aperiodic processes are similar to transformed periodic processes in that their execution time may span several slack intervals. If the aperiodic process has not completed when the slack has been consumed, it times-out and is suspended to later inquire whether more slack has become available.

Note that a higher criticality incoming aperiodic will preempt an executing aperiodic, slack updates will be made, and slack will be reallocated to the higher criticality aperiodic. Also observe that a requesting aperiodic with slack-level $i$ will only be allocated the slack available in the interval beginning at the time of request and ending at the completion time of the periodic with priority not greater than $i$ that would result in a missed deadline if the available slack were increased. Using time-outs in this way guarantees that an execution overrun in a process with criticality $j$ will not cause any processes with criticality greater than $j$ to miss their deadlines.

**Estimating Overheads and Errors**

Timing analyses of safety-critical systems must include estimates of kernel overheads, interprocess communication overheads and blocking times resulting from semaphore access in addition to the usual (typically worst case) process execution times. The MetaH toolset provides a sensitivity timing analysis tool which provides a feasibility analysis for MetaH specifications on a per processor per mode basis, and automatically includes the overheads mentioned for periodic processes. Overheads attributable to specific periodic processes such as blocking times and communication times are included as part of the periodic process’ compute times when making the initial slack calculations for the slack table. The kernel is modeled as a periodic process and automatically included in the analysis. Computation of the slack table is based on the periodic process’ execution times and all associated overheads (including the kernel process).

There is also kernel overhead associated with aperiodic processes. If there is interprocess communication between periodic and aperiodic processes, the calculation of slack becomes an iterative process where first the slack tables are produced using the periodic task set alone, then estimates of blocking times to periodic tasks resulting from the execution of aperiodic tasks is computed. These new estimates of periodic process compute times are then used to generate another slack table and the process continues until the sequences of slack table computations converge. Some difficulties in estimating even worst case blocking times introduced by aperiodics are mentioned in the last section.

Slack reallocation is inherent to the design of RMS, however the basic RMS algorithm does not keep track of how much slack is available, unlike the slack stealer. Knowing precisely how much slack is available at any
instant does come at some run-time expense. In addition to the slack table, there are sets of accumulators to keep track of how much aperiodic and idle execution time has been consumed and also how much slack has been reclaimed from periodic processes. These accumulators are updated at the end of every process execution, and in the case of a period transformed process, at the end of every process timeslice execution as well as when transitioning from an idle interval to a busy interval. Depending on the slack allocation policy, these accumulators may also need to be updated when an aperiodic process suspends itself or when an incoming aperiodic process preempts a currently executing aperiodic process. Slack accumulator updates are order \( n \), where \( n \) is the number of periodic tasks in an application. Since slack updates are relatively frequent and rely on clock measurements, it is important to include the slack update times as overheads attributed to processes (including the scheduler), and to assess the accuracy of the updates to determine whether clock measurement errors are either cumulative or large. Since slack accumulators are reset to zero at every hyperperiod, the error is at least bounded.

Updating slack accumulators is not the same operation as computing slack availability. However, the time spent updating the accumulators, the time spent computing slack and any measurement errors in actual execution times will be reflected in the accuracy of the value of the available slack. Available slack calculations are necessary in only two cases. If, when a process (periodic or aperiodic) either completes or suspends itself, there is an aperiodic process waiting with criticality greater than the next periodic process that would be scheduled at that time, available slack is computed for the waiting aperiodic process. The second case is when an aperiodic process with higher criticality than the one currently executing arrives. Once again, calculating the available slack is order \( n \), where \( n \) is the number of periodic processes. Slack need not be available, and in practice we require that the amount be greater than the time to make a context swap plus some for execution when allocated.

### Jitter and Table Size

A dynamic and mathematically equivalent calculation for slack has also been proposed that takes into account jitter and does not require on-line storage of a potentially large slack table[1]. This formulation is acknowledged by the authors as not being practical for many real-time systems, and an approximate algorithm is given which only occasionally computes the actual available slack dynamically. In this section, we discuss techniques for avoiding jitter in MetaH and some candidates algorithms for table size reductions.

In addition to any process jitter that might be present in a purely periodic system, slack stealing can alter the starting and completion times of periodic processes and thus may introduce additional jitter if caution is not exercised in design. In MetaH, jitter is minimized by separating the times at which processes input and output their signals and when they actually execute. Part of process dispatch is interprocess communication which is performed by the scheduler process. The scheduler process is the highest priority periodic process (and cannot be deferred to favor an event arrival by allocating slack), hence sampling times are deterministic to within a few microseconds.

One potential drawback that has been cited regarding the slack stealer is that very large table sizes might result[1]. It is possible to reduce table sizes at the expense of losing consumable slack. In the simplest and least optimal case, one would simply pick the smallest of all slack increments at each slack-level. This would result in a slack table with each row being constant, in which case only the \( n \) constants would be stored. When deadlines are equal to their periods and the periodic task set is harmonic, the jump heights increase by a constant at each slack-level. Consequently the slack table can be stored as a function, whose evaluation requires only a single multiplication. For convenience, many multi-rate control algorithms use harmonic rates. However, if preperiod deadlines or period transformations are supported (as they are in the MetaH slack scheduler), the functional representation need not hold. Alternative compressions techniques are possible, where a trade is made between the potential amount of usable slack and table size. Table reduction techniques must maintain a relationship between the slack table index and the dispatch instance (within the hyperperiod) if efficient slack table lookups are to be preserved.

### Areas for Future Work

Algorithms for scheduling slack as proposed in [1], [2] and [3] are quite general in their applicability to service arbitrary task streams in the context of RMS.
Because of this theoretical generality, the associated run-time implementations may experience large overheads without simplifying assumptions derived from the application domain. Areas in need of further investigation are estimating overheads, supporting mode changes for slack scheduled processes and simplifications of implementations resulting from assumptions about the aperiodic streams.

Estimating overheads introduced by slack scheduling of a nearly arbitrary stream (minimum interarrival times are implicitly determined by interrupt handling times) is difficult for many reasons. The size, location and even number of the slack increments vary depending on arrival times, execution times and interactions with other streams. So counting the worst case number of context swaps resulting from preemptions is not completely obvious. There is also the interaction between periodic and aperiodic processes mentioned previously that complicates this calculation. Additionally, estimates of the time to make the run-time slack variable updates are needed, which is actually incorporated in the run-time slack computation for slack availability. Large inaccuracies in these estimates could result in (difficult to detect) system faults by allocating slack that is not available.

Most safety-critical real-time systems require that some processes continue execution undisturbed across mode changes. MetaH supports this for periodic processes but not currently for aperiodic processes scheduled using the slack stealer. Difficulties arise since different modes will have different slack tables. It is possible to generate one slack table for all modes, but the size of such a table would almost certainly be prohibitive. Support for preserving aperiodic execution across mode transitions may require that mode changes occur only at hyperperiod boundaries, where the slack accumulators are all reset. Other restrictions to preserve process criticalities may be necessary.

One difference in our preliminary implementation and the slack stealer in [3] is that we currently do not provide on-line acceptance tests and guarantees of process completion for hard-deadline aperiodic processes. One case where on-line acceptance tests appear to be efficient is the incremental scheduling of periodic tasks. The task model represents each periodic task as a sequence of components, one of which is mandatory and the others optional. The set of all mandatory components is assumed to be a feasible task set and provide the minimal quality of service for each periodic task. The collection of optional components compete for the remaining slack. Periodic task $\tau_i$ then consists of $\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m_i}$.

Two distinctions exist between optional components and a completely general aperiodic task stream. First, since optional components have deadlines equal to their corresponding baseline component, the deadlines of $\tau_{i,j}$ and $\tau_{i,m_i}$, for $1 \leq j \leq m_i$ are all equal. When optional components are assigned the same criticality as their baseline component, the slack availability computation then returns a relatively quick on-line decision of whether the component will complete by its deadline. If the component is accepted, a quick update of the slack accumulators prior to executing the component will maintain the accuracy of on-line acceptance tests for subsequent component request (not necessarily from the same process sequence). Note that this on-line approach is similar to the approach in [3] in that an increment is accepted only if its deadline can be guaranteed. With increased overhead, it is possible to assign optional components different criticalities than their baseline components. Second, the arrival times of $\tau_{i,j}$ are not completely random. Specifically, $\tau_{i,j}$ would call await dispatch when and only if $\tau_{i,j}$ successfully completed, and $\tau_{i,m_i}$ would call await dispatch conditional on the successful completion of $\tau_{i,m_i}$, and this continues as long as each component successfully completes within the designated deadline.

References


