Memory-Aware Feedback Scheduling of Control Tasks [1]

Sven Gestegård Robertz
Department of Computer Science
Lund University
sven@cs.lth.se

Dan Henriksson   Anton Cervin
Department of Automatic Control
Lund University
{dan,anton}@control.lth.se

Abstract

This paper presents the incorporation of an auto-tuning real-time garbage collector into a feedback-based on-line scheduling system. The studied feedback scheduler is designed to dynamically adjust the sampling periods of a set of controller tasks in order to maximize the overall control performance. In the suggested approach, the memory management overhead is made explicit and taken into account by the feedback scheduler when scheduling the tasks. It is also described how priorities for memory allocations can be used to control the allocation rates of the application threads in order to optimize the trade-off between memory and CPU time usage. A case study, comparing theoretical analysis and simulated results support the feasibility of the approach.

1. Introduction

Real-time control systems are today often implemented using embedded microcomputers characterized by severe resource constraints in terms of, e.g., CPU and memory. In these resource-constrained environments, much can be gained in performance by using dynamic feedback-based resource allocation.

A characteristic property of feedback is that it can be used to reduce the effects of disturbances and to deal with uncertainties. The idea of *feedback scheduling* (FBS) is to use feedback to master uncertainty with respect to resource scheduling, such as, e.g., variations in task execution times. [12, 11] present strategies, where feedback is used to control the CPU utilization and deadline miss ratio of tasks with non-deterministic execution times and arrival patterns.

If the application tasks implement control algorithms, it is also necessary to take the control performance into account in the dynamic scheduling process. Controller tasks are generally not truely hard real-time tasks, but should rather be viewed as *adaptive*, in the sense that missing single deadlines does not jeopardize correct system behavior, but only leads to a performance degradation. Occasional deadline misses can be seen as disturbances acting on the control system.

Feedback scheduling schemes for control tasks designed to optimize the control performance are presented in, e.g., [7, 8]. These schemes adapt to changing requirements of the applications and tune the sampling periods of the tasks in order to keep the CPU utilization at a safe level while optimizing the quality of control. Feedback scheduling is also very suitable for systems which change between different operating modes [6] with different resource utilization patterns. For these control systems, designs based on worst-case assumptions would yield an unacceptably low CPU utilization and slow sampling. Thus, in the real-time systems community, the on-line resource management problem applied to CPU time has been thoroughly studied and the theoretical foundation of on-line scheduling is well built. Recently, with the development of processors with variable clock frequency, the relation between CPU time and power consumption has also been investigated [2]. Memory management, on the other hand, has been regarded as part of the application, and has not yet been considered in this context.

With the introduction of safe object-oriented languages with automatic memory management, like Java, the memory management overhead is moved from the application code to the run-time system. While viable in traditional systems, this introduction of automatic memory management complicates the picture when used in real-time systems. As the overhead of memory management is moved from application code to the run-time system, the isolation between tasks and the boundary between tasks and the run-time system is broken — there is still a cost of memory management, but it is no longer explicit. That is especially true for systems with a concurrent garbage collector (GC), an approach to real-time GC gaining in popularity [3, 16].

In this paper, we will study ways to incorporate the memory management problem within the existing framework for

[1]This work was financially supported by the Swedish Foundation for Strategic Research (SSF) and EU FP6, through the FLEXCON and SIARAS projects.

feedback scheduling of control tasks. This framework is based on the formulation and solution of a constrained optimization problem, where the performance is maximized subject to a utilization constraint [17]. First we will study how to make the cost of automatic memory management explicit in the period time optimization problem, so that it can be taken into account by the scheduler. We will then go on to present how to optimize the trade-off between memory usage and CPU time at run-time. The key problem here lies in the fact that both the CPU time and the memory are constrained resources, and that they cannot be managed independently.

1.1. Paper Outline

The remainder of the paper is organized as follows. Section 2 presents some preliminaries. Section 3 derives approximative models for estimating and optimizing the GC scheduling parameters together with the period assignment. Section 4 investigates how the mechanisms for different priorities on memory allocation requests presented in [13] can be utilized in a feedback scheduling context, where controlling the allocation rate of processes gives another degree of freedom when optimizing overall performance. This section also includes a case-study with analysis and simulation of the presented techniques applied to a control application. Finally, Section 5 concludes the paper.

2. Preliminaries

This section presents previous results in feedback scheduling and GC scheduling, on which the paper is based.

2.1. Feedback Scheduling

Feedback scheduling is an approach to handling temporal non-determinism, where the main goal is to optimize the resulting quality of service rather than aspects of the real-time scheduling, such as, for instance, the deadline miss-ratio. By using feedback control, the scheduling parameters are automatically adjusted at run-time in order to keep the CPU utilization at a safe level while optimizing the quality of service of the application [17, 1, 5, 7]. One area where this approach is useful is control systems, where it has been shown that the total quality of control can be dramatically increased if the real-time requirements are relaxed.

The structure of a basic feedback scheduler is as follows: A set of tasks generates jobs that are passed to a run-time dispatcher. The execution times of the jobs and the total CPU utilization, U, are assumed to be measured. Based on this, the scheduler adjusts the period times of the tasks, T_i, in order to keep the CPU utilization at the set-point, U_{sp}. (See also Figure 1.)

If a system contains both hard and soft real-time tasks, it is reasonable that the CPU utilization of the soft processes should be decreased more than that of the hard processes. This can be done by using *elastic scheduling* [4], where a stiffness value is assigned to each process and the scaling of period times is done in proportion to that value.

The general period assignment problem can be expressed as follows. A set of n tasks, $\tau_i, i \in \{1 \dots n\}$ with execution times C_i, adjustable periods h_i, and cost functions $J_i(h_i)$ share the same computer. The task of the feedback scheduler is to assign new sampling intervals $h_1 \dots h_n$ so that the global cost is minimized and the total CPU utilization is kept below a set-point, U_{sp}. This could be formulated as the optimization problem

$$\min_{h_1 \dots h_n} \sum_{i=1}^{n} J_i(h_i)$$
$$\text{subject to} \sum_{i=1}^{n} \frac{C_i}{h_i} \le U_{sp} \tag{1}$$

The cost functions, $J_i(h_i)$, measure the performance of the control loops, often expressed as a weighted quadratic sum of the control signal and the deviations of the controlled variables from the desired reference. The behavior of the individual cost functions depend on the complexity of the controllers and the controlled processes. However, linear or quadratic functions can often be used to approximate (or in some cases exactly model) the true cost functions.

If the cost functions, $J_i(h_i)$, are given by linear or quadratic functions, it has been shown [5] that a closed-form solution to the optimization problem (1) can be found. For example, with linear cost functions

$$J_i(h_i) = \alpha_i + \gamma_i h_i, \tag{2}$$

or, as functions of the sampling frequncies, f_i,

$$J_i(f_i) = \alpha_i + \frac{\gamma_i}{f_i}, \tag{3}$$

the optimal frequencies, $f_i^\star$, are given by

$$f_i^\star = \left(\frac{\gamma_i}{C_i}\right)^{\frac{1}{2}} \cdot \frac{U_{sp}}{\sum_{j=1}^{n}(C_j\gamma_j)^{\frac{1}{2}}}. \tag{4}$$

For the case of quadratic approximations of the cost functions, a similar explicit solution can be found [5].

2.2. Garbage Collection Scheduling

The discussion of GC scheduling is based on the *time-triggered GC scheduling* principle, where a concurrent GC is scheduled as a separate task, with an explicit deadline [16, 14]. Expressions for the GC cycle time have been derived.

Assume a set of n tasks, with frequencies f_i, and allocation requirements of a_i bytes per period; a heap size H, and a maximum total amount of live memory, L_{max}. It is then guaranteed that every GC cycle will be completed before the available memory is exhausted if the cycle time, T_{GC}, satisfies

$$T_{GC} \leq \frac{\frac{H-L_{max}}{2} - \sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} f_i \cdot a_i} \tag{5}$$

If *a priori* analysis is not available, on-line auto-tuning of T_{GC} can be done in a similar way, based on measuring the amount of available memory and the allocation rate. Let F be the amount of free memory and $\hat{\dot{a}}(t)$ be an estimate of an unknown but constant allocation rate, $\dot{a}$, such that $\hat{\dot{a}}(t) \geq \dot{a}$. During GC cycle k, with release time $R(k)$ and deadline $D(k)$ — i.e., for $R(k) \leq t < D(k)$, the GC cycle will be completed before the available memory is exhausted if the cycle time, T_{GC}, satisfies

$$\hat{T}_{\mathrm{GC}}(t) \leq \frac{1}{2}\left(t + \frac{F(t)}{\hat{\dot{a}}(t)} - R(k)\right) \tag{6}$$

Thus, previous work has studied how to calculate the scheduling parameters for a time-triggered garbage collector in two different cases. In the first one, all parameters (L_{max}, a_i, etc.) were known and constant. In the second case, the parameters were estimated based on run-time measurements. In a FBS system, the GC scheduling problem comes in a third form. Here, the parameters of the application (or *mutator*) tasks are known at any particular instant, but may change as the scheduler changes sampling rates in order to maximize the overall performance. Thus, there are dependencies between the GC and the feedback scheduler.

3. GC-aware Period Assignment

In a system with a scheduled garbage collector, the required CPU utilization of the GC, U_{GC}, must be taken into account when assigning task periods in order to keep the utilization below the setpoint. To get the total CPU utilization U_{sp}, the reference utilization *for the mutator tasks* in the feedback scheduler must therefore be reduced to

$$U_{ref} = U_{sp} - U_{\mathrm{GC}} \ . \tag{7}$$

The utilization of the garbage collector is $U_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}}{T_{\mathrm{GC}}}$ and thus, the constraint of the period assignment problem becomes

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{C_{\mathrm{GC}}}{T_{\mathrm{GC}}} \leq U_{sp} \ . \tag{8}$$

Given the previously derived expressions for the GC cycle and execution time, we get the the general expression for the required CPU utilization for GC,

$$U_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}(\textit{heap state})}{T_{\mathrm{GC}}(H, L, a_1, \ldots, a_n, h_1, \ldots, h_n)} . \tag{9}$$

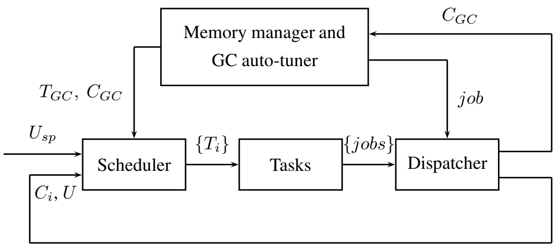

Figure 1. Feedback scheduling of both application tasks and GC. The GC task issues jobs which are dispatched just as any other jobs. The only difference between the GC task and the application tasks is that the GC is allowed to set its own period time while the feedback scheduler changes the application tasks' period times in order to keep $U \leq U_{sp}$.

However, at run-time, all parameters are typically not known, and therefore an approximate model must be used. We will now formulate such models for compensating for U_{GC} in FBS period assignment. In the first one, we will simply use a GC auto-tuner, as described in [16] and [14], as a reference generator to the feedback scheduler. In the second, we will incorporate the GC tuning into the optimization problem of the feedback scheduler, in the case where L_{max} is known. In the third one, we assume that L_{max} is unknown and derive similar expressions based on the previously described GC auto-tuning techniques.

As far as the optimization problem is concerned, we will assume that C_{GC} is constant. This just means that in the formulation of the optimization problem, we assume that C_{GC} is independent of the period times of mutator tasks, and that the effects that changes to the schedule have on C_{GC} are captured by the feedback loop. The interaction between the GC cycle parameter estimation and the feedback scheduling is done only through the model for T_{GC}.

3.1. Separate GC tuning and FBS

The most simple way of taking garbage collection work into account is to use the GC auto-tuner as a reference generator for the feedback scheduler. Figure 1 shows how the adaptive garbage collection scheduler fits into a general feedback scheduling system. The GC thread is scheduled as a normal application thread, but with the important difference that it is allowed to set its own deadline whereas the feedback scheduler changes the deadlines of the application threads in order to optimize CPU utilization.

As mentioned, the special treatment of the GC thread is necessary since the GC will stop all application threads if the system runs out of memory and that must be avoided as it leads to long GC pauses and unacceptable real-time performance. In this case, the GC tuner and the feedback scheduler are independent of each other, and the feedback scheduler simply uses U_{ref} as in (7), where T_{GC} and C_{GC} are estimated using some of the described techniques.

However, in general, the different tasks have different memory requirements, and thus any changes to the scheduling will affect the GC workload. As the GC scheduler is decoupled from the feedback scheduler, such effects cannot be taken into account in the period assignment, and this is a limitation of the described approach. Instead, any changes to the allocation rate — and, hence, to T_{GC} and U_{GC} — caused by the changes in period times are compensated for by the feedback to the GC tuner. That may, in turn, cause U_{ref} to change, and therefore, this model may show oscillating behaviour. Such oscillations can, however, be avoided by using conservative settings in the GC auto-tuner. For instance, if the U_{GC} prediction is filtered using the maximum value and a forgetting factor close to unity, a well damped system can be achieved, at the price of lower average utilization.

Another, and potentially more important, drawback of the separated approach is that the measured GC overhead is divided evenly across all mutator tasks. Thus, even if one task is responsible for the majority of the memory usage, the sampling rates of all tasks will be affected. In systems with competing (as opposed to cooperating) tasks, that may be an issue, as far as fairness in the scheduling is concerned.

3.2. Integrated GC and FBS

If the GC estimation and tuning is incorporated in the feedback scheduler itself, the effects on the GC utilization of changing period times can be taken into account in the period time optimization. In principle, we want to be able to express the cost of GC per task and sample, in a way that the constraint in the optimization problem is on a form that allows the existing closed-form solution (4) to be used.

Under the previously stated assumption that C_{GC} is constant, U_{GC} will be a function of the GC cycle time, which, in turn, depends on the allocation rate. Thus, we get a utilization constraint with one term for the CPU requirement and one for the memory requirement of each task,

$$\sum_{i=1}^{n} \frac{C_i + K_{\mathrm{GC}} \cdot a_i}{h_i} \leq U_{sp} \qquad (10)$$

where K_{GC} can be viewed as the cost, in CPU utilization, of memory allocation in CPU seconds per byte. With this formulation, the utilization constraint is in the same form as (1), as the extra term is constant (assuming a_i is independent of h_i), and thus the existing explicit solution to the optimization problem can be used. We will now see how the utilization constraint can be expressed when the maximum amount of live memory is known and unknown, respectively.

3.2.1 Using worst case live memory information

Given the maximum amount of live memory, L_{max}, and the amount of memory allocated per period of each task, a_i, we can use (5) to find the maximum allowed T_{GC} and, hence, the CPU utilization:

$$U_{\mathrm{GC}} = C_{\mathrm{GC}} \cdot \frac{\sum_{i=1}^{n} \frac{a_i}{h_i}}{\frac{H - L_{max}}{2} - \sum_{j=1}^{n} a_j} . \qquad (11)$$

Inserting (11) into (8) gives the constraint

$$\sum_{i=1}^{n} \frac{C_i + \frac{C_{\mathrm{GC}}}{\frac{H-L_{max}}{2} - \sum_{j=1}^{n} a_j} \cdot a_i}{h_i} \leq U_{sp} \qquad (12)$$

which, assuming that C_{GC} and $\{a_1 \ldots a_n\}$ are independent of $\{h_1 \ldots h_n\}$, can be written as (10).

In practice, the period time of the GC will be much longer than that of the mutator tasks, and thus $\sum_{j=1}^{n} a_j$ is typically very small compared to $H - L_{max}$. Further, if a conservative estimation of U_{GC} is used, and $U_{sp} < 1$, there will always be some slack in the schedule. For these reasons, sufficient safety margins can be achieved, making it reasonably safe to approximate (12) with

$$\sum_{i=1}^{n} \frac{C_i + \frac{C_{\mathrm{GC}}}{\frac{H-L_{max}}{2}} \cdot a_i}{h_i} \leq U_{sp} . \qquad (13)$$

I.e.,

$$K_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}}{\frac{H-L_{max}}{2}} \qquad (14)$$

which is precisely the GC CPU time per allocated byte.

3.2.2 Without a priori analysis

The above discussion assumes L_{max} to be known and that it is reasonable to use the worst case live memory. If that is not the case, T_{GC} can be estimated using (6), and the constraint (8) becomes

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{2\, C_{\mathrm{GC}}}{\frac{F(t)}{\sum_{i=1}^{n} \dot{a}_i} + t - R} \leq U_{sp} \qquad (15)$$

which, with $\dot{a}_i = \frac{a_i}{h_i}$, gives

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{2\, C_{\mathrm{GC}}}{\frac{F(t)}{\sum_{i=1}^{n} \frac{a_i}{h_i}} + t - R} \leq U_{sp} \qquad (16)$$

which can be reorganized as

$$\sum_{i=1}^{n} \frac{C_i + \frac{2\,C_{\mathrm{GC}}}{F(t)+(t-R)\sum_{i=1}^{n}\frac{a_i}{h_i}}\,a_i}{h_i} \le U_{sp}\ . \tag{17}$$

I.e.,

$$K_{\mathrm{GC}} = \frac{2\,C_{\mathrm{GC}}}{F(t)+(t-R)\sum_{i=1}^{n}\frac{a_i}{h_i}} \tag{18}$$

Unfortunately, the constraint (17) is not linear, meaning that the existing closed-form solution is not directly applicable. Worse yet, in this form, we get an optimization problem where both the objective function and the constraint are concave, and that makes it practically useless.

In order to remedy that, an approximation that turns (17) back into a linear constraint is sought. It is observed that, if $\dot{a}$ is constant, the denominator in (18) is equal to $F(R)$. If that is used to linearize the constraint, we get

$$\sum_{i=1}^{n} \frac{C_i + \frac{2\,C_{\mathrm{GC}}}{F(R)}\,a_i}{h_i} \le U_{sp} \tag{19}$$

and

$$K_{\mathrm{GC}} = \frac{2\,C_{\mathrm{GC}}}{F(R)}. \tag{20}$$

The error in the T_{GC} approximation of (19) will increase with increasing changes in $\dot{a}$ and the effect will be greater if the change occurs later in the GC cycle. Figure 2 shows how the approximation error depends on the change in $\dot{a}$ and the time of change. For instance, if the allocation rate is doubled half-way into the GC cycle, the relative error in the T_{GC} approximation will be 20 %. However, as the total GC utilization typically is 5–20 %, the overall impact of the error in the approximated utilization will only be a few percent. For robustness, a safety margin to accommodate such uncertainties can be added when setting U_{sp}.

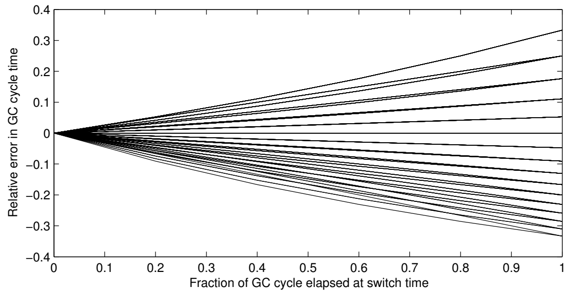

Figure 2. Relative error in T_{GC} approximation as function of change in $\dot{a}$ and time of switch. The lines represent changes in $\dot{a}$ from a factor of 0.5 to a factor of 2. An increase in $\dot{a}$ causes underestimation of U_{GC}.

3.3. Experiments

The performance of the two approaches was studied in a simulated environment. The set-up consisted of a system with two periodic tasks running under feedback scheduling. Figure 3 shows the reference utilization for the mutator (U_{ref}), and the sampling period of the first task (h_1), under both the separate and the integrated feedback scheduler. In all plots, the solid line represents the integrated scheduler, and the dashed line is the version with separate GC tuner. The integrated scheduler used the simplified constraint (20). As the approximation introduces errors, for a better comparison the resulting period times of the integrated scheduler were scaled to get a mutator utilization of exactly U_{ref}.

These experiments show similar performance for both the separate and integrated versions. That supports the claim that the most important difference is the fairness issue, as in the integrated version, the amount of allocation affects the period assignment. This is apparent in the right plot, where the difference in h_1 for the two schedulers is larger than in the other experiments, due to the bigger difference in memory usage. It can also be seen that as the GC utilization decreases, the variation in U_{ref} also decreases.

Thus, with suitable approximations, the CPU requirement of the GC task can be included in the period assignment, while keeping the optimization problem on a form that allows the existing closed-form solutions to be used.

3.4. Utilizing Slack

In order to get a system that is robust to variations in execution times, the utilization setpoint is typically set below 100 %. Also, to get stable estimates of GC scheduling parameters, the estimation needs to be conservative. That means that, in the average case, there will be some slack in the schedule, allowing the GC to finish before its deadline.

The feedback scheduler reserves a fraction of the CPU time for garbage collection. However, when the GC is not running, this CPU time could be used for mutator threads. In a system with a time-triggered GC, it is known that when the GC has finished a cycle it will not need to run again until at its next release time, $R(k+1) = D(k)$. If the feedback scheduler is aware of the state of the GC, this means that when the GC has completed a cycle, a higher mutator utilization can be allowed until the next GC release time. That is, if the GC finishes at time t_f ; $R < t_f < D$,

$$U_{ref}(t) = \begin{cases} U_{sp} - U_{\mathrm{GC}}, & R \le t \le t_f \\ U_{sp}, & t_f < t < D - \delta \end{cases} \tag{21}$$

where δ is used to take into account the fact that increasing the mutator utilization may increase the allocation rate and, hence, shorten the time until the next GC release.

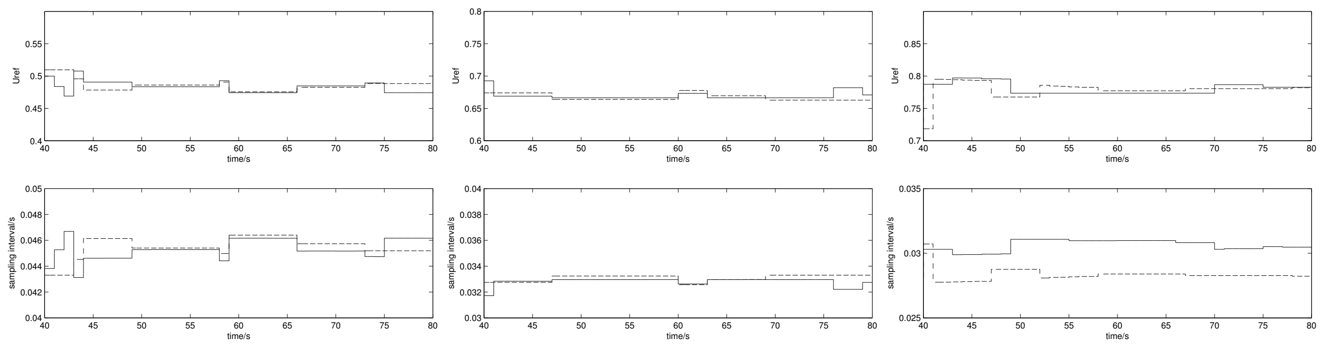

Figure 3. Comparison of separate and integrated scheduling, when heapsize (H), and allocation per sample (a_i) is varied. The plots show U_{ref} and h_1. Left: $a_1 = 300$, $a_2 = 640$, $H = 100000$, Middle:$a_1 = 300$, $a_2 = 640$, $H = 200000$, Right: $a_1 = 300$, $a_2 = 64$, $H = 200000$.

The GC cycle time, and consequently the release time of the next GC cycle, was estimated based on worst case assumptions about floating garbage, but when a GC cycle has finished, it is known how much memory was actually reclaimed. Thus, δ depends on both the amount of free memory and the allocation rate. We know that the amount of free memory at the time the GC has completed the cycle, $F(t_f) \geq F_{min}$. The requirement is the same; when the next GC cycle starts, the amount of free memory must be no less than F_{min}. Therefore, the adjusted release time of the next GC cycle must satisfy

$$R'_{\text{GC}}(\dot{a}) \leq t_f + \frac{F(t_f) - F_{min}}{\dot{a}} \tag{22}$$

and, with equality, we get

$$\delta = D - R'_{\text{GC}}(\dot{a}) = D - \left(t_f + \frac{F(t_f) - F_{min}}{\dot{a}}\right). \tag{23}$$

Thus, a sufficient degree of conservatism can be used to give robustness against inaccuracies in the GC scheduling parameter estimates due to variations and approximations, without the low average CPU utilization normally associated with such conservative scheduling.

4. Controlling the Allocation Rate

As we have seen, the fraction of CPU time that must be reserved for garbage collection depends on the allocation rate of the mutator, which, in turn, depends on the period times of the individual threads. Therefore, in a system with garbage collection, the FBS controls the CPU usage of a thread directly, through the period assignment, but also indirectly as the period time affects the allocation rate. If it were possible to directly control the allocation rates of the individual threads, that would increase the flexibility of a feedback scheduler, by making it possible to separate allocation of memory and CPU time. As higher memory usage means more GC work, that allows the scheduler, or resource manager, to trade off memory usage for CPU time.

In [13], it was observed that if not all parts of a hard real-time system are critical to the operation of the system, the non-critical parts of the system could be turned off in order to avoid overload. That led to the notion of priorities for memory allocations, where non-critical allocation requests can be denied by the run-time system if memory is scarce.

Assuming that each task has a critical and a non-critical part, with memory requirements of $a^{(c)}$ and $a^{(nc)}$, respectively, we extend the cost function with a term corresponding to the increase in quality from the non-critical parts

$$J(h, a^{(nc)}, \ldots) = \ldots \; ; \; 0 \leq a^{(nc)} \leq a^{(nc)}_{max} \tag{24}$$

which gives the optimization problem

$$\begin{array}{ll} \underset{h_1 \ldots h_n}{\min} & \sum_{i=1}^{n} J_i(h_i, a_i^{(nc)}, \ldots) \\ \text{subject to} & \sum_{i=1}^{n} \frac{C_i + K_{\text{GC}} \cdot \left(a_i^{(c)} + a_i^{(nc)}\right)}{h_i} \leq U_{sp} \end{array} \tag{25}$$

In the general formulation, (25) is expensive to solve online. In order to test the fundamental principle, we will now investigate a simplified case, where the problem is reduced to either allowing all or no non-critical allocations of a thread in each sample.

4.1 Case study: Ball-and-beam

Control of a ball-and-beam system will be used as an application example. This system consists of a horizontal beam, on which a ball should be balanced. The input to the system is a voltage controlling the angular velocity of the beam, and the measurements consist of the current

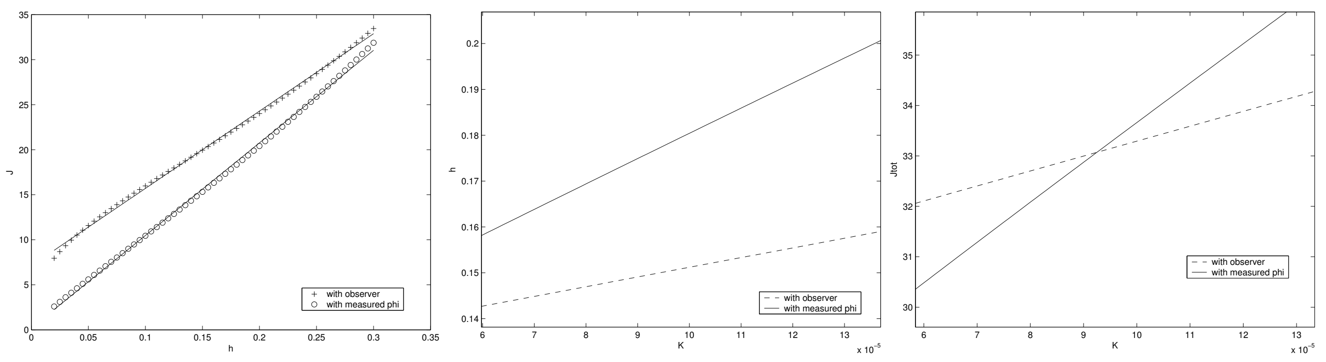

Figure 4. Left: The calculated costs and the linear approximations. Middle:Sample rate as function of K_{GC}. Right:Cost as function of K_{GC}. For high values of K_{GC}, the system with the observer will outperform the one with measured angle, due to the large CPU cost of memory allocation.

beam angle and ball position along the beam. The controller for this example is designed using linear-quadratic theory, where the objective is to minimize a quadratic sum of the control signal and deviations of the plant states. A Kalman filter is also designed to cope with the fact that not all states of the plant are available for measurement.

It is assumed that the beam angle can either be measured (which requires a measured-value object to be allocated and passed to the controller) or estimated using the Kalman filter. I.e., the allocation of the angle measurement is non-critical. Depending on the state of the memory system, K_{GC} — and hence, the total available CPU utilization — will vary.

Using Matlab-based tools, the effects of the scheduling on control performance in the described scenario is analysed and simulated. For control performance analysis, the Jitterbug toolbox is used. Jitterbug is a tool for studying how timing affects the performance of a computer-controlled system [10]. The simulation is done using the TrueTime [9] real-time kernel simulator in Matlab/Simulink, with simulated heap and GC [14, 15], the controller task, and one disturbance task.

Theoretical analysis

The analysis is done for two versions of the ball-and-beam controller: with or without angle measurements. The controller with the angle measurement will allocate more memory per sample, and therefore, under the discussed feedback scheduling, it will suffer a bigger penalty from the GC overhead. On the other hand, for the same sampling rate, the controller using angle measurements will perform better. In order to optimize quality of control, the cost of memory management must be balanced against the control performance, to choose which of the two controllers to use, given a certain K_{GC}.

Figure 4 shows the calculated total cost for a range of sampling rates, the sampling rate as a function of K_{GC}, and the resulting J as function of K_{GC} for the two systems (using linear cost functions). The controller without angle measurements has lower memory requirement, and is therefore much less sensitive to K_{GC}. The intersection of the lines in the right plot is the value of K_{GC} where the system with observed angle starts outperforming the one with measured angle as a lower memory usage allows a higher sampling rate — the optimal K_{switch}.

Simulation

In order to measure the control performance of the LQG regulator, if q_n is the weight of the nth state (i.e., $Q = \mathrm{diag}(q_1 \dots q_n)$), and x is the state vector, we define the total cost as

$$J_{tot} = \int^t \sum_{i=1}^{n} q_i x_i^2(t)\, dt\ . \qquad (26)$$

Running the system with different values of the switching point K_{switch} and measuring J_{tot} gives the plot shown in Figure 5, where the minimum corresponds to the optimal K_{switch}. The cost is the total cost of a $160s$ execution, and it is not normalized. The absolute values of the cost are not very interesting, as a direct comparison with the analysis is not possible as they show different things. The analysis calculated the cost for different, constant, values of K_{GC}. In the simulation, K_{GC} varied throughout the execution and at each scheduling instant the controller with the lowest cost was used. The GC was scheduled as described in [16], and the feedback scheduler used U_{GC} to adjust the utilization reference, according to Section 3.1.

In theory, the minimum in Figure 5 should be at the same K_{GC} value as the intersection of the lines in the right part of Figure 4. The discrepancy between the theoretical and

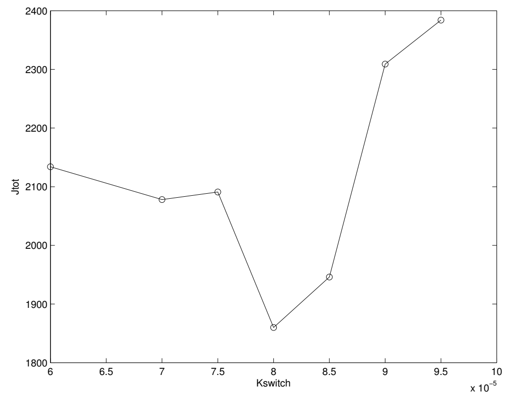

Figure 5. Total cost as function of K_{switch}.

simulated results can be explained by a combination of inaccuracies in both the models and the run-time system. The theoretical results are based on an optimal feedback scheduler, but at run-time, some approximations are required. Notably, in order to determine the mutator utilization, both the GC cycle time and the GC execution time has to be predicted. The GC cycle time is dependent on the allocation rate and object distribution, which are both affected by the mode changes. Also, in order to get a high enough K_{GC} to reach K_{switch}, the system had to be stressed by increasing C_{GC} in the simulator, resulting in a U_{GC} around $45 - 55\ \%$. Thus the impact of the discussed approximations and uncertainties, which would be small in a system with lower U_{GC}, became significant.

While the setup in this simple case study is not entirely realistic, it still illustrates the fundamental idea that if memory usage can be controlled, the total quality of control of a system can be improved by on-line optimization of the trade-off between memory and CPU usage.

5. Conclusions

This paper has presented extensions to feedback-based on-line scheduling of control tasks, where the effects of an auto-tuning real-time garbage collector has been taken into consideration. Approximate models were developed for estimating and optimizing the GC scheduling parameters. These models were then used as added constraints when optimizing the sampling periods of the control tasks. The use of priorities for memory allocation was also investigated in the same context. The methods are supported by evaluation in a simulated control application case study.

References

[1] L. Abeni and L. Palopoli. On adaptive control techniques in real-time resource allocation. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. on Computers*, 53(5):584–600, 2004.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL'03*, New Orleans, Louisiana, USA, Jan. 2003.

[4] G. Buttazzo, G. Lipari, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3), Mar. 2002.

[5] A. Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, Apr. 2003.

[6] A. Cervin and J. Eker. Feedback scheduling of control tasks. In *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, Dec. 2000.

[7] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1), July 2002.

[8] D. Henriksson and A. Cervin. Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC 2005*, Seville, Spain, Dec. 2005.

[9] D. Henriksson, A. Cervin, and M. Andersson. TrueTime – simulation of networked and embedded control systems. http://www.control.lth.se/user/dan/truetime, 2006.

[10] B. Lincoln and A. Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proc. of the 41st IEEE Conference on Decision and Control*, Las Vegas, 2002.

[11] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-time Systems*, 23(1/2):85–126, 2002.

[12] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, 1999.

[13] S. G. Robertz. Applying priorities to memory allocation. In *Proc. of the 2002 International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, June 2002. ACM Press.

[14] S. G. Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Department of Computer Science, Lund University, 2006.

[15] S. G. Robertz. Simulating a garbage collected heap in Matlab/TrueTime. Technical report, Dept. of Computer Science, Lund University, 2006.

[16] S. G. Robertz and R. Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Langauges, Compilers, and Tools for Embedded Systems – 2003 (LCTES'03)*, San Diego, California, USA, June 2003. ACM Press.

[17] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, USA, 1996.