Feedback Control Static Scheduling for Real-Time Distributed Embedded Systems

Tolga Ayav
INRIA Rhône-Alpes
ZIRST 655, Avenue de l'Europe
38334 Montbonnot, St Ismier Cedex, France
Tolga.Ayav@inrialpes.fr

Yves Sorel
INRIA, Domaine de Voluceau
Rocquencourt - B.P.105, 78153
Le Chesnay Cedex, France
Yves.Sorel@inria.fr

Abstract

This paper presents an implementation of feedback control strategy on distributed static scheduling. The static schedule is created taking into account the average execution times of the tasks. Feedback control algorithm handles the unestimated dynamic behaviors in the system and keeps the performance at a desired level. The approach of feedback control supporting static scheduling yields more flexible scheduling, low scheduling overhead and better resource utilization while preserving the real-time constraints.

1. Introduction

Scheduling for distributed embedded systems is becoming more important. Scheduling of tasks on a distributed architecture is a resource allocation problem and there are two kinds of allocation policy: static and dynamic. The dynamic policy is more efficient and its implementation is also simpler. However, the disadvantage is the overhead both in program size and in execution time. Static scheduling minimizes the overall execution time drastically reducing overheads but all the properties of the application, including its environment must be known at compile time. Therefore, static policy is appropriate for the implementation of real-time control, signal and image processing algorithms on embedded systems, where resources and time are hardly limited and where the algorithm and its environment are well-known.

Today, most of the distributed embedded systems operate in open environments where both workload and available resources are difficult to predict such as mobile vehicles, robots and so on [3][5]. We present a feedback control distributed static scheduling (FC-DSS) system that combines feedback control scheduling and static scheduling. The proposed approach results in lower overhead and better resource utilization, which makes it attractive for embedded systems with limited resources.

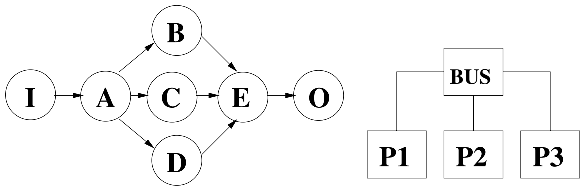

Figure 1. Example algorithm graph (left panel) and architecture graph (right panel).

2. Feedback controlled static scheduling framework

We use the AAA heuristic [2] to obtain the static schedule of the algorithm application. AAA is based on the complete knowledge of task execution times. When it is difficult to define the workload and execution times, systems may not meet the deadline. To overcome this problem, we propose to use average execution times instead of worst case and an additional feedback control loop to handle this unestimated characteristic of the system.

2.1. Algorithm, architecture and task model

The algorithm specifying the application to be implemented is modeled by a data-flow graph (denoted with $\mathcal{G}_{al}$). Each vertex represents a task and each directed edge represents a data-dependence. The data-flow graph is executed periodically and each execution is called an iteration. An example of algorithm graph is given in Figure 1. The period of the iteration is the deadline for the completion time of all the tasks. This hard real-time constraint is difficult to meet when there exists a lack of prior information about the execution times.

The architecture is modeled by a graph (denoted with $\mathcal{G}_{ar}$) where each vertex is a processor or a communication medium (bus, link, etc.) and each non-directed edge is a connection between a processor and a medium. Processors (rep. media) execute sequentially computations (resp. communications) tasks (see Figure 1).

2.2. Static scheduling heuristic

We briefly present the static scheduling heuristic implementing this solution. Let $O_{sched}(n)$ denote the list of already scheduled tasks and $O_{cand}(n)$ denote the list of candidate tasks built from the algorithm graph at step $n \geq 0$. A task is candidate when all its predecessors are already scheduled. Initially, $O_{sched}(0) = \emptyset$. Using a cost function called *schedule pressure*, one operation is chosen to be scheduled at step n. The heuristics computes first the "the earliest start date from start" $S(n)(\tau_i, p_j)$ for each task τ_i and processor $p_{j\in\{1,2,\cdots,n\}}$. $S(n)(\tau_i, p_j)$ takes into account the communication times between τ_i and its successors and predecessors when they are distributed on different processors. Thus the schedule pressure σ is computed as:

$$\sigma(n)(\tau_i, p_j) = S(n)(\tau_i, p_j) + ET_{i1,pj} + E(\tau_i) - R \quad (1)$$

where $ET_{i1,pj}$ is the execution time of τ_i on processor p_j; this value is given in p_j's characteristics lookup table. Note that among k versions of τ_i, the longest version is utilized in the heuristics. At each step n, one task is selected and implemented on a processor until all tasks are scheduled according to the following algorithm:

```
$S_0$. Initialize $O_{sched}$ and $O_{cand}$:
  $O_{sched}(0) = \emptyset, O_{cand}(0) = \{\tau \in O | pred(o) \subseteq O_{sched}(0)\}$
$S_n$. while $O_{cand}(n) \neq \emptyset$ do
  $S_{n_1}$ Compute the scheduling pressure for each
      $\tau_i \in O_{cand}(n)$ and keep the result
      for each operation:
      $\sigma^{opt}(n)(\tau_i, p_i) = \min_{p_j \in P} \sigma(n)(\tau_i, p_j)$
      $P(\tau_i) = p_i$
  $S_{n_2}$ Select the best candidate $\tau_{best}(n)$ such that:
      $best = \arg(\max_{\tau_i \in O_{impl}(n)} \sigma^{opt}(n)(\tau_i, p_{i_l}))$
  $S_{n_3}$ Implement $\tau_{best}$ on $P(\tau_{best})$
  $S_{n_4}$ Update list of candidates and scheduled tasks:
      $O_{sched}(n) = O_{sched}(n-1) \cup \{\tau_i\}$
      $O_{cand}(n+1) = O_{cand}(n) - \{\tau_i\}$
                  $\cup\{\tau_i^{'} \in succ(\tau_i) | pred(\tau_i^{'}) \subseteq O_{sched}(n)\}$
  end while
```

For further detail about AAA, one may refer to [2].

2.3. Imprecise computations revisited

Imprecise computations allow programmer to identify some tasks as more important than others. It includes a number of implementation techniques such as milestone, sieve, and multiple-version that are presented thoroughly in [4]. In the present work, multiple-version implementation is considered. This means that each task is composed of many versions such as primary, secondary and so on. The primary one is the longest version that produces a precise result, whereas the other versions are shorter and they produce imprecise results.

We propose an imprecise computation which assumes that each task $\tau_i \in \mathcal{G}_{al}$ is described with a tuple {I, ET}. Each task τ_i has k versions, i.e. $I = \{\tau_{i1}, \tau_{i2}, \cdots \tau_{ik}\}$. Since the target architecture is heterogeneous, the execution times for a given task can be distinct on each processor. Thus, each version has different execution times on n different processors, i.e. $ET = \{\{ET_{i1,p1}, ..., ET_{i1,pn}\}, ..., \{ET_{ik,p1}, ..., ET_{ik,pn}\}\}$ (with the assumption that $ET_{i1} > ET_{i2} > ... > ET_{ik}$). Note that these execution times are average instead of worst case execution times in order to achieve higher CPU utilizations in the system. Feedback control loop provides a satisfactory performance even when ETs vary from their estimated average values during run-time.

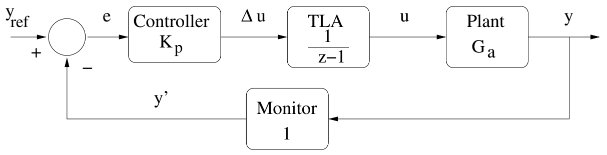

Figure 2. Feedback control loop

2.4. Feedback control loop

We insert a feedback control algorithm into the application algorithm. A model of the loop is given in Figure 2. The control loop is invoked once at each iteration of the global schedule. The controlled variable denoted with y is the normalized completion time of the whole algorithm, i.e. the completion time divided by the period of the iteration T. For the rest of the paper, the term completion time should be interpreted as normalized completion time. The reference variable y_{ref} is the desired value of y. We implement this feedback control loop with the algorithm graph $\mathcal{G}_{fc}$ given in Figure 3. Thus, our proposed model extends the whole algorithm to be scheduled to $\mathcal{G}_{al} \cup \mathcal{G}_{fc}$. We explain comprehensively *Monitor*, *Controller* and *TLA* parts below.

2.4.1. Monitor

The monitor measures the completion time of the whole algorithm. For any processor, let us denote the completion time of the last task in one iteration with C_i, and then normalize it with $m_i = \frac{C_i}{T}$. Here, m differs from CPU utilization with the fact that CPU idle time when tasks wait for data from their predecessors is also included in m calculation. The reason of using m as the controlled variable is that

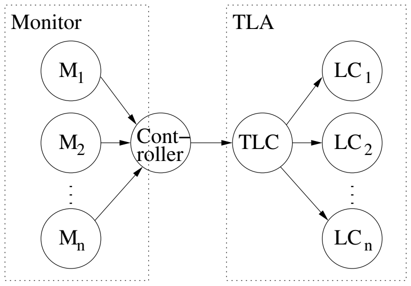

Figure 3. The graph of the feedback control loop algorithm ($\mathcal{G}_{fc}$)

the end of completion time of the algorithm is not allowed to exceed period T. Using m instead of CPU utilization as the controlled variable has also another advantage that CPU utilization cannot exceed 100% whereas m can. Therefore we do not have the difficulties of non-linear analysis and a limitation on y_{ref} (for more explanation, please refer to [3]).

```
Monitor{
    for $i = 1$ to $n$
        $m_i(t) = Receive\_Data\_From(M_i)$;
    end for
    $y'(t) = \max\{m_i(t)\}$;
}
```

where $t \in \mathbb{N}$ denotes the discrete time index, i.e. the actual time $\tilde{t}$ can be computed as $\tilde{t}$=Tt.

2.4.2. Controller

We use P (Proportional) control function to calculate the control output. The reason of using a simple P controller is that its implementation introduces less overhead, and as already remarked in [5] and [3], a P control structure is enough flexible to stabilize the system.

Controller first calculates the error, executes P control action and calls TLA for the selected processor. The pseudo-code of the controller is given below:

```
Controller{
    $e(t) = y_{ref} - y'(t)$;
    if($e(t) < 0$)
    then proc=Select_CPU_having_highest_m_and
                    _minimum_one_degradable_task();
    else proc=Select_CPU_having_highest_m_and
                    _minimum_one_enhancable_task();
    $\Delta u(t) = K_p \cdot e(t)$;
    $TLA(proc,\ \Delta u(t))$;
}
```

Δu is the output of the P control action. $\Delta u > 0$ means that the requested completion time of the algorithm should be increased, and $\Delta u < 0$ means that the requested completion time of the algorithm should be decreased. The controller then calls task level actuator (TLA) to change the current requested completion time from u to $u + \Delta u$. In other words, negative error means that the completion time of the algorithm exceeds the desired value and some tasks must be switched to their secondary versions so as to decrease the completion time. In this case, controller selects the processor having the highest completion time (i.e. the processor that defines the completion time of the whole algorithm) and minimum one degradable task. Then, it executes the controller and TLA for that processor. Degradable means that task's version can be changed from primary to secondary so as to decrease the completion time. If the error is positive, then the controller selects the processor having the highest completion time and minimum one enhancable task in this case.

The schedule of the whole algorithm is given in Figure 4. Since $y' = 1.01 > y_{ref}$ and there may exist unpredictability in the execution times of the tasks, the feedback control loop dynamically tries to keep y at the desired level specified with y_{ref}.

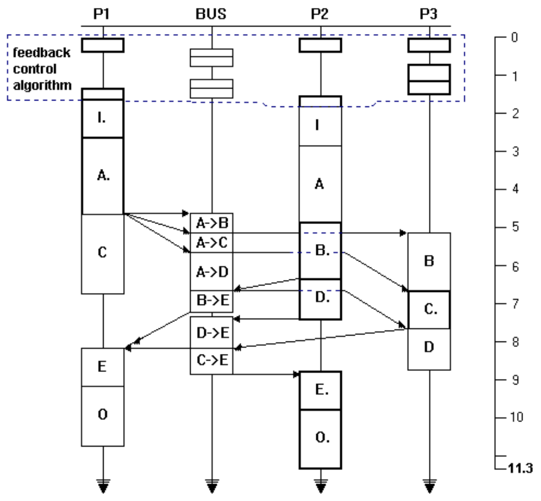

Figure 4. Final schedule

2.4.3. Task level actuator

The task level actuator (TLA) manipulates the requested completion time of the algorithm by changing the versions of the tasks. For instance, if it changes the version of task τ_i from τ_{ip} to τ_{ir}, it adjusts the requested completion time by calculating $ET_{ir} - ET_{ip}$. TLA, then sends the information about the chosen versions of the tasks to the LC_i (*Level Controller*) tasks that adjust the versions to be executed in the next iteration. The pseudo-code of TLA is as follows:

```
$TLA(proc,\ \Delta u)\{$
  if($\Delta u < 0$)
```

```
        while($\Delta u < 0$ and $If\_Degradable\_Task(proc)$){
            $\tau_c = Select\_Degraded\_Task(proc)$;
            $Change\_Level(\tau_c$, from version p, to version r);
            $\Delta u = \Delta u - ET_{ir,proc} + ET_{ip,proc}$;
        }
    else
        while($\Delta u > 0$ and $If\_Enhancable\_Task(proc)$){
            $\tau_c = Select\_Enhanced\_Task(proc)$;
            $Change\_Level(\tau_c$, from version p, to version r);
            $\Delta u = \Delta u - ET_{ir,proc} + ET_{ip,proc}$;
        }
    for $i$=1 to $n$
            $Send\_Data\_To(LC_i)$
    end for
}
```

2.5. Control theory based analysis

Designing a stabilizing controller is an important concern [3][1]. In stability analysis, we first derive the state space form of the closed-loop system given in Figure 2. Starting from the control input, TLA changes the estimated completion time of the algorithm for the next period at every sampling instant t such that $u(t+1) = u(t) + \Delta u(t)$. Since the precise execution time of each task is unknown and time varying, the actual completion time (denoted with y) may differ from the estimated completion time $u(t)$ such that $y(t) = G \cdot u(t)$ where G represents the maximum extent of workload variation in terms of requested completion time. We assume that the monitor measures the actual completion time precisely, i.e. $y^{'}(t) = y(t)$.

Then, we derive the error and the control signals as $e(t) = y_{ref} - y^{'}(t)$ and $\Delta u(t) = K_p e(t)$ respectively. The closed-loop system is therefore described by

$$u(t+1) = (1 - K_p G)u(t) + K_p y_{ref}. \qquad (2)$$

A classical stability criterion for discrete time linear systems guarantees that $u(t)$ is asymptotically stable around its equilibrium, if the following condition is fulfilled:

$$|1 - K_p G| < 1 \Longleftrightarrow 0 < K_p < 2/G \qquad (3)$$

Assuming that the stability condition 3 is satisfied, control system guarantees zero steady state error, i.e.

$$\lim_{t\to\infty} e(t) = y_{ref} - G \lim_{t\to\infty} u(t) = y_{ref} - G\frac{y_{ref}}{G} = 0. \qquad (4)$$

3. Simulations

In the experiments, we used the example application algorithm and architecture given in Figure 1. The execution times of primary versions of the tasks and the communication times are generated randomly. The secondary versions are the half of the primary versions. Due to the space limitation, these values cannot be given here. The controller parameter K_p and the reference variable y_{ref} are assigned 0.5 and 0.85 respectively. In order to account for the unpredictability of these execution times, the actual execution times are calculated as $AET_{pi} = ET_{pi} \cdot uniform[0.8, 1.2]$ which are unknown to the feedback control scheduler. Figure 5 shows the completion times of the three processors during 50 iterations. As seen in the figure, feedback control algorithm keeps the completion time around y_{ref}.

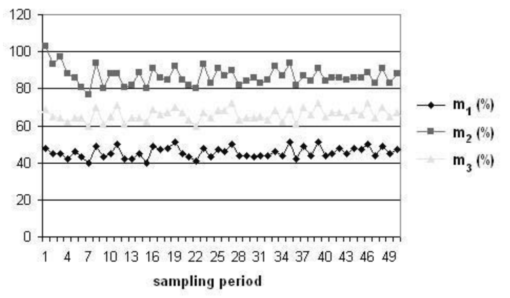

Figure 5. Task completion times ($y_{ref} = 0.85$).

4. Conclusion

In this paper, we presented a feedback control static scheduling technique for distributed embedded systems. Static schedule is created in accordance with the AAA heuristic and an additional feedback control algorithm handles the possible unestimated behaviors in the workload. We provide a stability analysis to tune the parameters. The results are demonstrated through a simple experiment. The proposed scheduling system should also be tested under more realistic workloads.

References

[1] T. Ayav, G. Ferrari-Trecate, and S. Yilmaz. Stability properties of adaptive real-time feedback scheduling: A statistical approach. In *12th Real-Time Embedded Systems Conference*, pages 259–277, Paris, France, 30-31 March 2004.

[2] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, Italy, May 1999.

[3] C. Lu, J. Stankovic, G.T., and S. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems Journal Special Issue on Control Theoretical Approach to Real-Time Computing*, 23(1/2):85–126, September 2002.

[4] W. Shih and J.-S. Liu. Algorithms for scheduling imprecise computations. *IEEE Transactions on Computers*, 24(5):58–68, 1991.

[5] J. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. In 11th *Euromicro Conference on Real-Time Systems*, pages 11–20, York, UK, 1999.