Simulation Results and Algorithm Details for Value Based Overload Handling

Jan Carlson, Tomas Lennvall and Gerhard Fohler
Department of Computer Engineering,
Mälardalen University, Sweden
{jcn,tlv,gfr}@mdh.se

Abstract

In this paper we present the simulation results for a proposed algorithm for value based task rejection in the presence of offline scheduled tasks for which a timely execution have to be guaranteed. We also describe in detail the algorithm for computing overload amounts.

1 Algorithm for computing overload amount

Given the deadlines and remaining execution times of the aperiodic tasks, and the spare capacity (slots not reserved for offline scheduled tasks) of consecutive intervals, this algorithm computes the overload amount of each aperiodic task.

Let $\tau_1 \ldots \tau_n$ be a sequence of aperiodic tasks sorted by increasing deadline. Also, assume a sequence of consecutive, non-empty, time intervals, each associated to a number of offline scheduled tasks as defined by the slotshifting algorithm [Foh95]. The following additional notation is used in the algorithm.

dl_x	the deadline of τ_x
c_x	the remaining execution time of τ_x
end_x	the end time of interval number x
sc_x	the spare capacity of interval number x
oa_x	will be assigned the overload amount of τ_x

Algorithm

Let ct be the current time, and ci the number of the interval that ct belongs to. Further, assign $oa_1 := c_1$. If the algorithm is called with compute-oa$(ct, 1, ci, sc_{ci})$, then oa contains the overload values for τ_1 to τ_n, upon termination.

function compute-oa(t, d, i, c)
if $d \leq n$ **then**
  if $dl_d < end_i$ **then**
    $tmp := \min(c, dl_d - t)$
    $oa_d := oa_d - tmp$
    if $d < n$ **then** $oa_{d+1} := oa_d + c_{d+1}$
    compute-oa$(dl_d, d + 1, i, c - tmp)$
  else
    $oa_d := oa_d - c$
    compute-oa$(end_i, d, i + 1, sc_{i+1})$

Note that the function is tail-recursive and thus can be implemented with bounded memory, e.g., as a standard imperative loop.

Complexity

Before considering the complexity of the algorithm, we formulate an invariant, i.e., a proposition that is true every time the function is called. For this, we define $in(x)$ to be the number of the interval containing the time x. This allow us to formulate the invariant as $i \leq in(dl_d)$.

The correctness of the invariant is proven as follows. For the initial call to the function, we have $i = ci \leq in(dl_1)$ since no task in the sequens has already violated its deadline. Next, we assume that the invariant holds for one call, and show that this implies that it must hold for the next recursive call as well.

If the first branch of the if-then-else statement is selected, i is unchanged and $d$is increased by one in the next recursive call. Since $in(dl_d) < in(dl_{d+1})$, and since $i \leq in(dl_d)$ by assumption, we have $i \leq in(dl_{d+1})$ so the invariant holds for the next call as well.

If, instead, the else branch is selected, we must have $dl_d \geq end_i$. Assume further that the invariant does not hold for the next call. Than, since it holds for the current call, we must have $i = in(dl_d)$. This implies that $end_i \geq dl_i$, which leads to a contradiction and thus proves that the invariant must hold for the next call.

By induction, we have now shown that the invariant holds each time the function is called.

Since we have $d \leq n$, the invariant implies $i \leq in(dl_n)$. Also, we know that d and i are never decreased, that one of them is increased in each recursive call, and that they are initalised to 1 and ci respectivly. This implies that the total number of calls to the function can be no more than $n + m$, where m is the number of intervals between the current time, and the deadline of τ_n. Thus, the worst case time complexity of the algorithm is in $O(n + m)$.

2 Simulations

We have implemented the algorithms described in [CLF03], and have simulated various scenarios. The simulated system consists of 8 processing nodes, connected via a network where all necessary messages can be sent during one time slot.

Each simulation has a length of 2000 slots. The randomly created offline schedules have a load of 0.4, evenly distributed over the nodes, and their length varies between 300 and 1000 slots.

Worst case computation time for both offline and aperiodic tasks varies uniformly in the range 1–10. Aperiodic tasks are assigned an actual execution time uniformly distributed between 0.5 and 1.0 of its wcet, and relative deadlines varying between 1–3 times wcet.

Arrival times of aperiodic tasks are distributed over the simulation length, with the restriction that no task have a deadline exceeding the simulation length. Finally, values of aperiodic tasks vary uniformly in the range 1–100.

The total system load varies between 0.8 and 3.0, the offline load of 0.4 included. The load parameter is based on wcet, and thus represents the load as perceived by the overload algorithm. The actual system load is lower[1], since execution time is less than wcet.

[1] The actual system load varies approximately between 0.7 and 2.35 in the experiments, calculated from the distribution of actual execution times

Experiment 1: Method comparison

We have studied the total accumulated value of aperiodic tasks that finished in time, and the following methods have been compared:

1. The full method presented in the paper (*Migration*).
2. The overload handling algorithm, without task migration (*Local*).
3. A basic algorithm that uses the offline schedule, assigning idle slots to aperiodic tasks based on value density (*Offline Valuedensity*).
4. Same as 3, but aperiodic tasks are ordered by value (*Offline Value*).
5. Same as 3, but aperiodic tasks are ordered EDF (*Offline EDF*).
6. Same as 3, but aperiodic tasks are serviced in order of arrival. (*Offline FCFS*).

Methods 1 and 2 implement the efficiency improvements suggested in [CLF03]. Each point in the figures represents some 300 simulations.

In the first part of the experiment, all nodes in the system are subject to the same amount of load. The result is presented in Figure 1. Here, the possibility of task migration does not provide any significant improvement. Compared to the basic method, the performance of the proposed method is significantly higher.

The second part of the experiment, shown in Figure 2, is a scenario of unevenly distributed load. Half of the nodes have no aperiodic tasks arriving, only offline scheduled tasks. Here, the task migration algorithm clearly increases the system performance, compared to overload handling without migration, because tasks can migrate to nodes with no aperiodic load.

Experiment 2: Restrictions

The theoretical worst case time complexity of the overload algorithm, for a ready queue of length n, is $O(n^2)$. This experiment shows how the execution time is affected by system load, and the impact on performance from restricting the algorithm as suggested in [CLF03] to deal with complexity issues.

The parameter *cutoff* denotes the maximum length of the ready queue. I.e., tasks that are inserted at a position greater than *cutoff* are automatically rejected, which means that they are placed in the maybe-later queue (if they just arrived, or if they were in the ready queue during the previous slot), or not stolen (if they were from a maybe-later queue).

We have measured the total accumulated value of aperiodic tasks that finished in time (similar to experiment 1) for different *cutoff* values. Execution time has been approximated by the number of arithmetic, comparison and assignment operation performed in the overload algorithm, including the computation of σ-values.

The parameters are the same as in experiment 1, with the load evenly distributed over the nodes, and using the full method from the paper (*Migration*). Figure 3 shows the accumulated value for different *cutoff* values. In Figure 4, the average number of operations for a single call to the overload algorithm is presented. Figure 5 gives the maximum number of operations performed during a single call to the overload algorithm. Each point in the figures represents some 300 simulations. Thus, in figures 4 and 5, each point represents over 4 million calls to the overload algorithm (8 nodes, and a simulation length of 2000).

In practice, the execution time is not as big an issue as the theoretical complexity suggests. None of the 57 million calls to the overload algorithm made during simulations needed more than 720 operations to be performed.

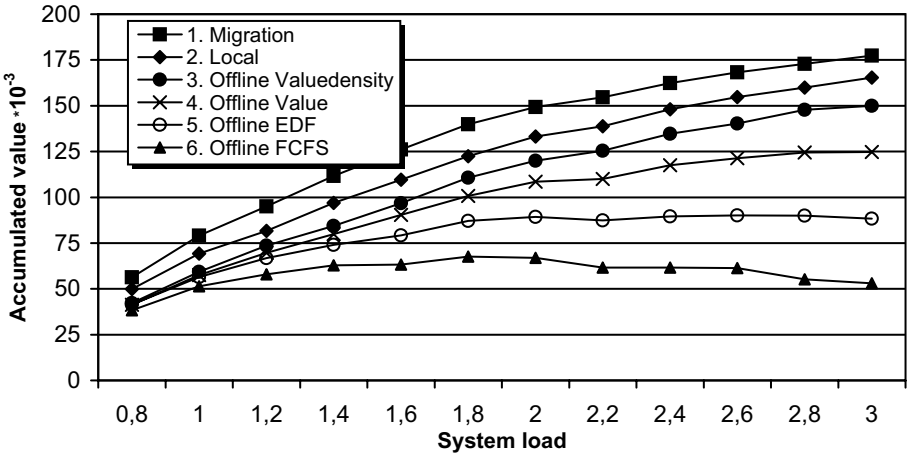

Figure 1: Accumulated value for even load distribution.

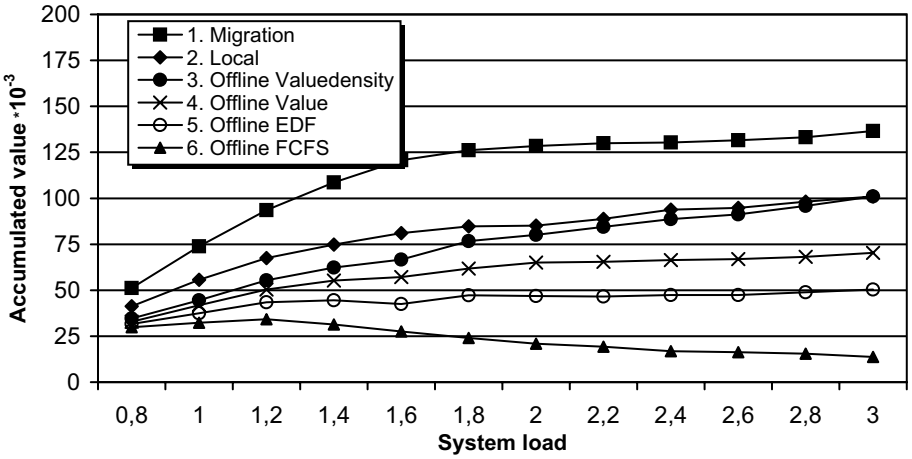

Figure 2: Accumulated value for uneven load distribution.

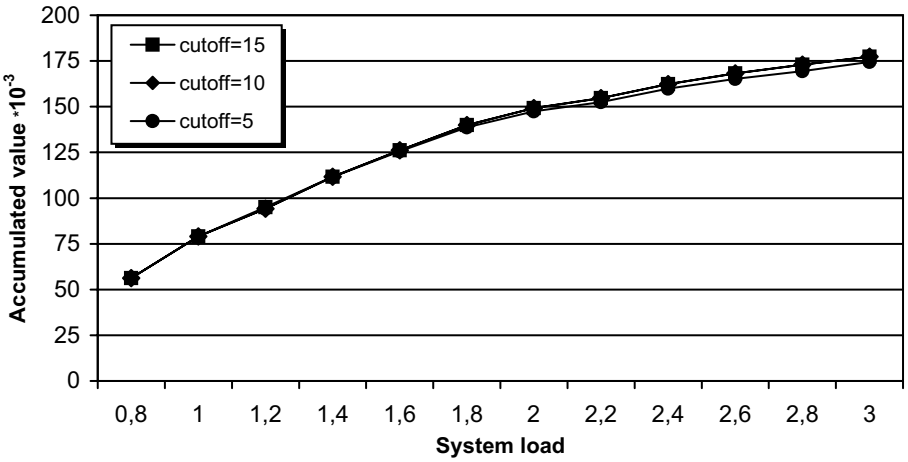

Figure 3: Accumulated value for different *cutoff* values.

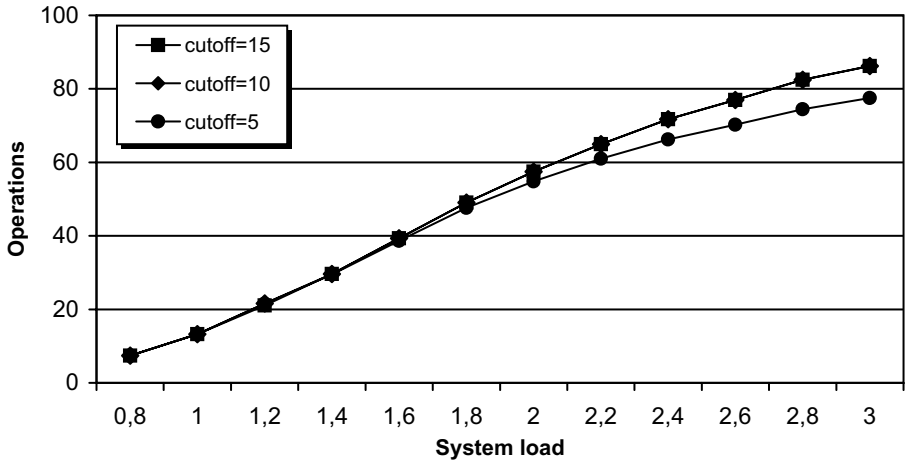

Figure 4: Average number of operations for different *cutoff* values.

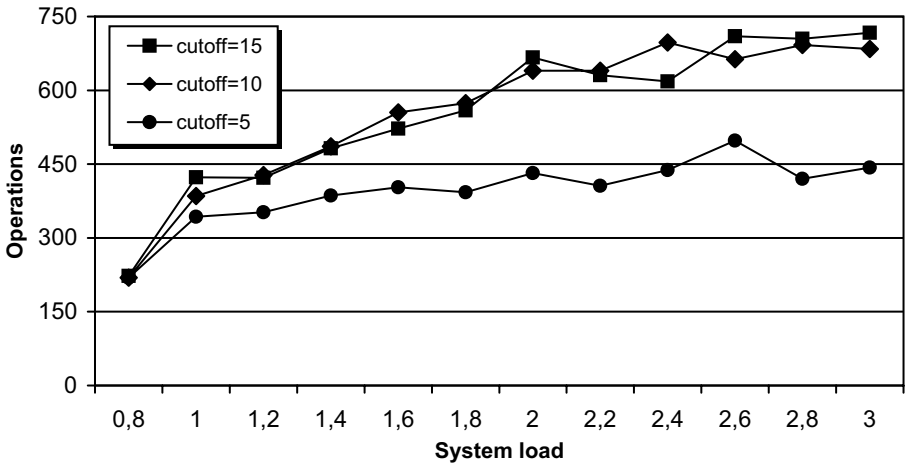

Figure 5: Maximum number of operations for different *cutoff* values.

This is partly because the ready queue size (which is the parameter used in the complexity analysis) is not proportional to system load. Also, the worst case assumes that none of the restrictions are trivially solved by the solution to the previous ones, which is highly unlikely when the queue is long.

The simulations show that restricting the length of the ready queue significantly reduces worst case execution time, with only a moderate performance decrease.

References

[CLF03] J. Carlson, T. Lennvall, and G. Fohler. Enhancing time triggered scheduling with value based overload handling and task migration. In *6th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Hakodate, Japan, May 2003.

[Foh95] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, Dec. 1995.