Scheduling Transient Overload with the TAFT Scheduler

M. Gergeleit and E. Nett

Institute for Distributed Systems (IVS)
Otto-von-Guericke Universität Magdeburg
{gergeleit, nett}@ivs.cs.uni-magdeburg.de

EXTENDED ABSTRACT

Nearly all known real-time scheduling approaches rely on the knowledge of Worst Case Execution Times (WCETs) for all tasks of the system. This is a severe restriction, as with today's complex systems it becomes increasingly more expensive to determine WCET by applying analytical methods and the computed results often turn out to be a very pessimistic upper bound for the observed execution time. Moreover, in many complex real-time applications that consist of heterogeneous components it is virtually impossible to compute WCETs at all.

Also, more and more research is undertaken to enable real-time systems to cope at least with *transient overload* [Ber01, Ric01]. Transient overload occurs, whenever the system needs more computing resource than available in order to be able to complete all tasks before their deadline. With the classic concept of WCETs, transient overload can be avoided by an appropriate acceptance test that the scheduler uses to check schedulabilty before a task set is admitted to execution. However, even in the world of hard real-time systems with known WCETs it has turned out, that sometimes the overall load imposed by the environment can be higher that expected (either because of wrong design assumptions or dynamic changes in the system or its environment). If this happens, it is unacceptable for a safety-critical system to fail completely. Instead some kind of graceful degradation is desirable, where the system tries to ensure the timely execution of at least the most *important* tasks.

In both cases, unknown WCETs and transient overload, the assumption that all parameters of a system are known a-priori does not hold any more. This lead us to the idea of using estimated execution times that are based on statistics and system monitoring instead of fixed WCETs. However, this also means that these estimated execution times can turn out to be wrong for some (longer-lasting) task instances and a scheduler that has based its decisions on these wrong assumptions has to deal with this situation. One approach to achieve this is to trade optimal timeliness for functionality, a well-known strategy in fault tolerance. A timing fault occurs whenever the actual execution time of a task is greater than expected, either caused by a wrong timing estimate or by transient overload. The TAFT (*Time-Aware Fault-Tolerant*) scheduling [Net01] is able to handle these kinds of timing faults in a way such that deadlines are still met.

This presentation first sketches TAFT scheduling. Then, it discusses a scheduling scheme and an execution environment that makes use of a combination of *Earliest Deadline* scheduling strategies to implement TAFT. This scheme allows for a simple acceptance test and achieves a small scheduling overhead at run-time. Finally, simulation results are presented that show the achieved real-time behavior with an increased acceptance rate, a higher throughput, and a graceful degradation in transient overload situations compared to standard schedulers.

1 Overview of the TAFT Scheduling

In TAFT each task is designed as a TaskPair (TP) with a common deadline. A TP constitutes a MainPart (MP) and an ExceptionPart (EP), thus reflecting the mentioned fault-tolerance aspect. The MP contains the real application code, the EP, as the name suggests, contains the exception handler that is executed in case of a timing error of the MP. More formally, TAFT applications are constituted by a set $\prod$ of independent tasks τ_i designed as a TP, which is a tuple with deadline D_i composed by a MP and an EP. The EP execution time (E_i) is characterized as *Worst-Case Execution Time* (WCET), while in the MP its execution time (C_i) can be interpreted as *Expected-Case Execution Time* (ECET)

[Ger01]. Each TP is also characterized by an activation time T_i - its period in the assumed case of periodic tasks. In summary, TAFT applications can be denoted as follows:

$$\prod = \{\tau_i = (T_i, D_i, C_i, E_i),\ i = 1 \text{ to } n\}.$$

Informally speaking, the ECET associated with the MPs is a measure for the time that a certain percentage of instances of a task needs for a successful completion [Net01]. Let t be an instance of a periodic task τ:

$C_{t,\,\alpha}$ = the CPU-time that has to be assigned to instance t in order get a probability α that t is completed.

More formally, $C_{t,\,\alpha}$ is the α-quantile of the probabilistic density function (PDF) that denotes τ's execution time. In most cases ECETs are considerably shorter than WCETs and it is assumed that $C_i + E_i << \text{WCET}_{\text{MP}}$. This leads to feasible schedules even in cases where a traditional WCET based scheduler will not find one. By assigning different values for α to different tasks, a notion of importance can be expressed. The closer α gets to 1 the more resources are allocated for a task by the scheduler to ensure a high probability that the MP completes before the deadline.

ECETs can be obtained from a dynamic system by means of online monitoring, the *Time Awareness* component of TAFT. With the use of ECETs as guaranteed time quantum for MPs the computational progress of a system can be preserved in the long term, as a predefined fraction α of τ's executions will be able to complete successfully. As most of the EPs are not executed at all there is even more CPU-time (than the guaranteed ECET) available for the MPs to complete in time.

From the scheduler's point of view, both parts are treated as separate scheduling entities having their individual timing parameters. The TAFT scheduler is a hard real-time scheduler in the sense that it always guarantees the completion of either the MP or the EP before D_i, thus reflecting the fault-tolerance aspect. An EP has to be executed if and only if its MP cannot be successfully completed before the latest point in time when the EP can be started without violating D_i. Executing the EP implies the abort of the respective MP. The application functionality of the TP is implemented in the MP. The functionality of the EP is minimal, and serves to ensure that the respective TP leaves: (*a*) the controlled application in a fail-safe state; and (*b*) the controlling system in a consistent state. Therefore, the assumption that the E_i is known is not a severe restriction, as it is assumed to be a simple and short piece of code. Thus, even bad and very pessimistic execution time estimation for the EPs should not lead to resource requirements that are comparable with those of the MP (i.e. $C_i >> E_i$).

2 Adopted Scheduling Strategy

A two-level dynamic scheduling strategy is adopted to schedule the tasks designed according to the TAFT approach described in the previous section. The first level is in charge of scheduling the EPs and the second one is responsible for the MPs. The main reason for using separated scheduling policies is that MPs and EPs can be viewed as distinct tasks that have different eligibility policies. On one side there is the MP, which must terminate before the EP release time (L_i). There are no restrictions related to its start time. On the other hand, EPs have a severe constraint regarding their start times (L_i) that, as previously mentioned, should occur as late as possible in order to maximize the execution time available for the MP.

The initial idea was to schedule the EPs in a calendar-based manner. However, implementing calendar-based algorithms presents many problems. The most obvious one is that a system based on this approach is brittle: It is relatively difficult and CPU consuming to modify and maintain dynamically. This has a direct consequence for the acceptance test: guaranteeing the timely execution of tasks in such an approach means finding an available time slot in the calendar or, in other words, calculating a complete schedule. Moreover, this has to be done not only when new tasks arrive, but also when a periodic task that has already executed should be re-planned. Depending on the relation among the periods of the existing tasks, a plan for the largest period may contain lots of entries (consuming lots of memory). Therefore, a simpler scheduling strategy is needed to make TAFT scheduling applicable in common real-time execution environments.

2.1 Scheduling EPs

The *Latest Release Time* (LRT) algorithm [Liu00], or *Earliest Deadline Least* (EDL) [Che89], was employed to schedule EPs, representing the first-level scheduler. Basically this algorithm can be regarded as "reverse EDF", treating release times as deadlines and deadlines as release times and scheduling the jobs backwards, starting from the task set's super-period P (the *least common multiple* - LCM - period) and going until the instant t where the tasks are simultaneously

released. This property imposes the use of a periodic task set. It also suggests that the scheduler has exactly the same configuration at time $t \geq 0$ that it does at time $t + kP$ ($k = 1, 2, \ldots$). So, finding the form of the schedule produced over infinite length by LRT amounts to find the form of the schedule on the intervals $[kP, (k + 1)P]$, $k = 1, 2, \ldots$, each of them being called a window.

Starting from P, the schedule is produced in a priority-driven manner, where the priorities are based on the release times of the tasks: the later the release time, the higher the priority. The LRT algorithm is proven to be optimal under the same conditions that the *Earliest Deadline First* (EDF) algorithm is optimal [Liu73]: it will feasibly schedule a periodic, preemptive task set with deadlines equal to periods, as long as its utilization factor is below 1.

2.2 Scheduling MPs

Considering the MPs, the adopted scheduling policy is EDF, hence assuming, that periods are equal to deadlines ($D_i = T_i$). One of the reasons is because the goal when scheduling MPs is to enhance CPU utilization, as EDF is optimal regarding this aspect. Another reason is that it complies with the scheduling strategy adopted for the EPs (reverse-EDF). Therefore, the proposed scheduling strategy can be formally defined as a monoprocessor scheduler, for a set $\prod$ of independent tasks τ_i designed as a TP, with deadline D_i equal to the period T_i, ECET C_i for the MP, and WCET E_i for the EP. The LRT algorithm schedules the EPs in the first-level, while the EDF schedule the MPs in the second-level. The priorities generated as output of these two schedulers are divided in two separate classes, similar to the approach proposed in [Dav95]. All priorities generated by the LRT are higher than those of the EDF. This means, that if there is at least one EP ready to run, it will immediately preempt any MP. Only if there are no EPs ready, the dispatcher chooses the MP with the highest priority assigned by the EDF.

However, with the scheduling and dispatching scheme as described so far, the guarantee for the availability of CPU-time for other MPs becomes void, as soon as one MP exceeds its ECET. A pure dispatching based on the priorities of EDF will result in the well known *Domino Effect*, where all subsequent tasks suffer from the fact that one task was running late. As the MPs' schedule is based only on ECETs and not on WCETs, by definition this is expected to happen for a certain percentage of tasks instances and is not at all an error. Therefore, our dispatcher knows about the ECET of the MPs and accounts for their time quantum already used. When an MP exceeds its ECET, its priority is decreased to a level that is below all EDF-assigned priorities of other MPs. This prevents it from consuming further CPU-time that is needed by EDF to execute the remaining MPs. Only if there are no other guaranteed task ready for execution, the MP can keep using the CPU (in order to complete before the latest release time of its EP). Again, it has to be emphasized that the execution of the EPs does not depend on this mechanism and is guaranteed independently due to the higher priorities assigned by the LRT.

2.3 Acceptance Test

While TAFT does not rely on WCETs, it is still not at all a best-effort scheduler. A feasible schedule of the TAFT-scheduler guarantees that each TP receives an execution time of at least C_i for the MP and, if required, E_i for the EP. This still holds in situations of transient overload (assuming that the overload affects the MPs execution time). In this case it still assigns the guaranteed execution time to each TP. However, the exception rate may go up, meaning that C_i is actually not enough time to ensure the desired success rate for the MPs.

In order to check in advance, whether a feasible schedule can be constructed by the applied LRT/EDF scheduling strategy, an acceptance test is required. So far a simple and sufficient acceptance test has been developed and proven for harmonic task sets. The developed acceptance test consists in calculating a so-called *maximum utilization factor* (MUF), as defined bellow.

Definition 1: Let the set $\prod = \{\tau_i = (T_i, D_i, C_i, E_i), i = 1 \text{ to } n, \forall i,j: (T_i \leq T_j \Rightarrow T_i \mid T_j)\}$ be ordered by crescent deadlines, which are equal to the task period. We define the *maximum utilization factor* (MUF) Ω_i for each TP τ_i as:

$$\Omega_i = \sum_{j=1}^{i} \frac{C_j + E_j}{T_j} + \frac{1}{T_i} \sum_{i<j\leq n} E_j$$

The MUF value represents a relative processor utilization index, which is defined as the expected CPU utilization when the TP executes within its *critical period* (CP) [Che89]. A TP is said to be within its CP when it is in the last activation before reaching the LCM (super-period) of the task set. Using this MUF the acceptance test for a task set is given by the following theorem:

Theorem 1: A task set $\prod$ of TPs (from definifion 1) is schedulable with the TAFT scheduler (level 1: EPs with LRT, level 2: MPs with EDF) if $\Omega_i \leq 1$ for each τ_i.

A similar (and simple) acceptance test for the more general case of non-harmonic task sets is a current topic of research.

3 The Execution Environment

For implementing the scheduling strategy described above the RT-Linux operating system [Bar97] on a Pentium CPU was used. The main reasons for choosing RT-Linux was that it already has a well-suited dispatching mechanism (strictly priority based, like in the majority of the available real-time OS), and that it is an open-source program, allowing modifications to be done right in the original scheduler. The implementation of the proposed execution environment consists in adding the previously described two-level scheduling scheme, including a mechanism to abort MPs, and enhancing the dispatcher to account for the actual CPU-time consumed by each task instance.

In order to guarantee the scheduler activation for the abort of a MP, a timer is programmed for the instant when the EP should be triggered (L_i). Considering the jitter to treat an MP's abort, the worst case is equal to the size of the operating system tick, which is 6.3 µs in the current implementation. At this moment, if the corresponding MP has not finished, the EP will be forward to the dispatcher. EP instances are automatically removed from the LRT queue when the respective MP terminates successfully, i.e. without violating L_i. Each time a periodic task has its next activation reprogrammed, a new instance of the EP is inserted dynamically into the LRT-scheduler.

To implement the acceptance test and the scheduler itself, it was necessary to place additional information in the task descriptor structure originally defined in the RT-Linux environment: L_i, C_i, and E_i. In addition, the scheduler needs to keep updated information about the CPU utilization (based in the information given by the active task set). This information is used in the acceptance test for the new tasks.

One of the more sophisticated implementation issues was the mechanism to abort a MP and to trigger the corresponding EP as soon as a deadline violation is detected. Two issues have to be considered:

1. The current instance of the MP has to be aborted, not just preempted. As a consequence, the thread executing the MP has to be set to a state that allows it to restart the MP from the very beginning in the next period.
2. While it would be easy to implement the EP in a real separate thread, this is not desirable. Exception handling is much easier if local state from the MP is still available. This is the usual semantics of language-level exception. Also, from the scheduler's point of view, it is not required to maintain two separate threads, as by definition the execution of the MP and the EP of one TP are mutual exclusive.

The obvious approach to fulfill these requirements is to use a kind of *setjump()*, *longjump()* mechanism as known from Unix/C. This requires saving an initial state of the MP and, in case of an exception, putting the aborted MP back in that state and to initiate the EP from there. In contrast to the Unix/C mechanism, the *longjump()* operation is actually invoked by the scheduler, not by the executing thread itself. The *setjump()* mechanism is implemented by saving the stack pointer, and the *longjump()* accordingly by restoring the stack state. For implementation reasons this restoring is performed in the context of a helper-thread, if the currently running MP is the one to be aborted, as it is quite difficult to manipulate the stack of a thread while being inside its own context.

4 Evaluation

The experiments conducted for experimentally evaluating the behavior and the efficiency of the presented scheduling strategy, especially in case of high load and overload situations, are constituted by a task set and a load similar to the *Hartstone* benchmark [Wei92]. In order to explore the properties of the TAFT scheduler, some modifications in the execution times from the baseline Hartstone benchmark are introduced: all tasks are partitioned into a MP and an EP, attributing the originally proposed execution time to the MP and additional 5% from this time to the EP. Furthermore, as the proposed TAFT-scheduling scheme is intended for dealing with variations in the execution times (reason why ECET are used), a PDF is used to model different execution time values for each instance of a periodic task. For the tests we assume that execution times range from 50% to 100% of the WCET. This assumption is conservative in a sense that the difference between worst-case and average-case tends to be larger and that the theoretical WCET is hardly ever reached, especially

on modern computer architectures. With this assumption the ECET C_i is defined as the α-quantile (e.g. the 0.95-quantile) of the used PDF. For the experiments we have selected the beta and the uniform PDF. As the average execution times are now only a fraction of the WCET compared to the Hartstone benchmark, also the overall utilization of the CPU is only a fraction of the value that is reached when all task compute up to their WCET. The overall utilization of the base task set tests is 70% of the WCET-case for the beta(2, 3), and 75% for the uniform PDF. This means, that the task set with the maximum load that passes the EDF acceptance test, still leaves more than 25% of the CPU unused. However, any attempt to handle more than this load with EDF (e.g. in case of unpredictable transient overload), leads to unpredictable timing behavior and deadline misses. In contrast, with TAFT there are some EP activations at an overall utilization of 85% (remember, based on the WCET-calculation this is already about 120% load), but still, as required, 90% of all MPs are completed successfully. Above 95% utilization there is a graceful degradation, but up to 105% load all tasks achieve an MP success ratio better than 50%. It can be observed, that even at nearly 130 % utilization the overall number of successful MPs is much higher than the number of timely terminations in the EDF-case and that no TP ever exceeds its deadline.

Due to the EDF scheduling policy for MPs, using TAFT also implies that the tasks with the higher periods (and the longer execution time) are penalized. In overload situations they run first into very high rates of EP activations. This is undesirable, as a high period not necessarily means low importance, but a simple priority-based scheduler has no means to express importance independently of the dispatching priority. Things are different in TAFT: by choosing different α-quantiles in the C_i parameters of the tasks, the desired average MP success-rate can be adjusted orthogonal to its period or its overall execution time. Experiments show, that the α-quantile can be used to express the importance of a task. The α-quantile in the ECET of an MP controls the expected completion rate. In overload situations a higher α-quantile that represents a higher importance leads to a decreased rate of ET-activations.

5 References

[Bar97] Barabanov, M. *A Linux-based Real-Time Operating System*. M.S. Thesis, New Mexico Institute of Technology, June, 1997.

[Ber01] Bernat, G. and Cayssials, R., "Guaranteed On-Line Weakly-Hard Real-Time Systems". In Proc. of 22nd IEEE Real-Time Systems Symposium. London, UK, 2001. pp. 25-35.

[Che89] Chetto, H. and Chetto, M. "Some Results of the Earliest Deadline Scheduling Algorithm". *IEEE Trans. on Software Engineering*, 15 (10), 1261-1269, 1989.

[Dav95] Davis, R. and Wellings, A. "Dual priority scheduling". *In Proc. of 16th IEEE Real-Time Systems Symposium*. San Francisco, US, 1995. pp. 100–109.

[Ger01] M. Gergeleit. A Monitoring-based Approach to Object-Oriented Real-Time Computing, Dissertationsschrift, Otto-von-Guericke-Universität Magdeburg , Universitätsbibliothek, 2001, http://diglib.uni-magdeburg.de/Dissertationen/2001/margergeleit.pdf

[Liu73] Liu, C.L., and Layand, J.W. *Scheduling Algorithms for Multiprograming in a Hard-Real-Time Environment*. Journal of the Assoc. for Computer Machinery 20 (1).

[Liu00] Liu, J.W.-S. *Real-Time Systems*. Prentice-Hall, 2000. ISBN 0-13-099651-3.

[Net01] E. Nett, M. Gergeleit, and M. Mock. "Enhancing O-O Middleware to become Time-Aware", *Special Issue on Real-Time Middleware in Real-Time Systems*, 20 (2): 211-228, March, 2001, Kluwer Academic Publishers. ISSN- 0922-6443

[Ric01] Richardson, P., Sieh, L., and Elkateeb, A., "Fault-Tolerant Adaptive Scheduling for Embedded Real-Time Systems", IEEE Micro, Sept/Oct 2001. pp. 41-51.

[Wei92] N. Weiderman and N. Kamenoff. Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems. *Journal of Real-Time Systems*, 4, 1992.