EPiC Series in Computing
Volume 63, 2019, Pages 251–261
Proceedings of 32nd International Conference on Computer Applications in Industry and Engineering

A Software Architecture for Handling Complex Critical Section Constraints on Multiprocessors in a Fault-Tolerant Real-Time Embedded System

Jia Xu

Department of Electrical Engineering and Computer Science
York University, Toronto, Canada
jxu@cse.yorku.ca

Abstract

In a real-time embedded system which uses a primary and an alternate for each real-time task to achieve fault tolerance, there is a need to allow both primaries and alternates to have critical sections/segments in which shared data structures can be read and updated while guaranteeing that the execution of any part of one critical section will not be interleaved with or overlap with the execution of any part of a critical section belonging to some other primary or alternate which reads and writes on those shared data structures. In this paper a software architecture is presented which effectively handles critical section constraints where both primaries and alternates may have critical sections which can either overrun or underrun, while still guaranteeing that all primaries or alternates that do not overrun will always meet their deadlines while keeping the shared data in a consistent state on a multiprocessor in a fault tolerant real-time embedded system.

1 Introduction

It is highly desirable to be able to effectively handle complex critical section constraints on multiprocessors in a fault tolerant real-time embedded system which uses a primary and an alternate for each real-time task to achieve fault tolerance. A *fault tolerant design* is often necessary to enable a safety-critical system, such as an aircraft or automobile control system to continue to provide a specified service, possibly at a reduced level of performance, rather than failing completely, in spite of system errors such as program/software errors due to software bugs having occurred or occurring. One approach for achieving fault tolerance in real-time embedded systems, is to provide two versions of programs for each real-time task: a *primary* and an *alternate*. If an error in the execution of the primary of a task is detected, or if the successful completion of the primary cannot be guaranteed, then the alternate will be activated, while the primary will be aborted [3-7]. In any kind of real-time embedded system, it is very common for concurrent real-time tasks/processes to need to read and update shared data resources, such as common data structures in shared memory that are shared with other concurrent real-time tasks/processes. The software architecture presented in this paper effectively handles the complex constraints in the execution of critical sections which read and update shared data structures in both primaries and alternates on a multiprocessor in a fault tolerant real-time embedded system. Many embedded systems applications have hard timing requirements where real-time tasks/processes with complex critical section constraints must be completed before specified deadlines. This requires that the worst-case computation times of both primaries and alternates of the real-time tasks be estimated with sufficient precision during system design, which sometimes can be difficult in practice. If the actual computation time of a primary or an alternate during run-time exceeds the estimated worst-case computation time, an *overrun* will occur, which may cause the primary or alternate to not only miss its own deadline, but also cause a cascade of other primaries and alternates to also miss their deadline, possibly resulting in total system failure. However, if the actual computation time of a primary or an alternate during run-time is less than the estimated worst-case computation time, an *underrun* will occur, which may result in under-utilization of system resources. The software architecture also allows each critical section in both primaries and alternates to either overrun or underrun while guaranteeing that all primaries or alternates that do not overrun will always meet their deadlines. The software architecture effectively utilizes any additional processor capacity created at run-time due to primary or alternate underruns to significantly increase the chances that either the primary or the alternate of each real-time task will be able to successfully complete the correct execution of its critical sections before its deadline despite overrunning.

Work by other authors related to using primaries and alternates in a real-time system include [3-7], while work by other authors related to handling underruns and overruns include [8-10, 13]. To the author's knowledge, none of the earlier work in [3-10, 13] allow both primaries and alternates to have critical sections which can either overrun or underrun, while still guaranteeing that all primaries or alternates that do not overrun will always meet their deadlines while always keeping the shared data in the critical sections in a consistent state. A significant contribution of the work presented in this paper, is that this is the first time that a software architecture has been presented which can effectively handle critical section constraints where both primaries and alternates may have critical sections which can either overrun or underrun, while still guaranteeing that all primaries or alternates that do not overrun will always meet their deadlines while keeping the shared data in a consistent state. The software architecture significantly increases the chances that either the primary or the alternate of each real-time task will be able to successfully complete its computation before its deadline despite overrunning, which significantly increases system robustness and reliability. None of the earlier work, including other authors' work such as [3-10, 13], and this author's work [11][12][14][15], have done this.

2 The Pre-Run-Time Phase of the Software Architecture

The steps of the pre-run-time phase are as follows.

(1) Provide two versions of sequential programs for each real-time task/process: a primary and an alternate. Organize the sequential programs as a set of periodic processes to be scheduled before run-time. Each periodic process p can be described as a quintuple $(o_p, r_p, c_p, d_p, prd_p)$. prd_p is the *period*. c_p is the worst case *computation time* required by process p. d_p is the *deadline* of process p. r_p is the *release time* of process p. o_p is the *offset*, i.e., the duration of the time interval between the beginning of the first period and time 0.

Each process p then also consists of two parts: a *primary* p_P and an *alternate* p_A.
A *latest start time LS(p_P) for each primary* p_P, and a *latest start time LS(p_A) for each alternate* p_A is determined before and during run-time.

For each process p, the primary p_P is executed first, and if the primary p_P is able to successfully complete without fault on or before reaching the latest start time LS(p_A) of the corresponding alternate p_A, then the corresponding alternate p_A will *not* be executed.

An alternate p_A will be *activated* and executed only if the corresponding primary p_P faults, or if the corresponding primary p_P is *not* able to successfully complete without fault on or before reaching the latest start time LS(p_A) of the corresponding alternate p_A, in which case the primary p_P will be *aborted*.

(2) Divide each primary or alternate into process segments such that appropriate exclusion and precedence relations can be defined on pairs of sequences of the process segments to prevent simultaneous access to shared resources and ensure proper execution order. If a segment x in any primary or alternate *PRECEDES* segment y in any other primary or alternate, then segment y cannot start execution before segment x has completed its computation. If a segment x in any primary or alternate *EXCLUDES* segment y in any other primary or alternate, then the execution of segment x cannot interleave or overlap with the execution of segment y. Exclusion relations can be used to prevent primaries and alternates from simultaneously accessing shared resources such as shared memory [11][12][14][15].

(3) Compute a feasible pre-run-time schedule S_O on a multiprocessor for all the segments in the primaries and alternates of processes which satisfies a given set of "EXCLUDES" and "PRECEDENCE" relations defined on ordered pairs of process segments in the set of periodic processes P, by applying the method in [15].

(4) Given any feasible pre-run-time schedule S_O on a multiprocessor, a set of "PREC" relations on ordered pairs of process segments in the set of periodic processes P in the feasible pre-run-time schedule S_O is defined as follows:

For all segments x, y:
if $e(x) < e(y) \wedge ((x \text{ EXCLUDES } y) \vee (x \text{ PRECEDES } y))$
then let x PREC y

(5) Given a feasible pre-run-time schedule S_O on a multiprocessor which satisfies a given set of "EXCLUDES" and "PRECEDENCE" relations defined on ordered pairs of process segments in the set of periodic processes P, one can use the procedures described in [14] or [15] to compute before or during run-time a *"latest-start-time schedule"* S_L, and *"latest start times"* for all the periodic processes in P, which can also be used to compute before or during run-time a "latest-start-time schedule", and "latest start times" for all the primaries and alternates in that set of processes, while maintaining the order defined in the "PREC" relations in the original feasible pre-run-time schedule S_O, such that all the "EXCLUDES" and "PRECEDENCE" relations defined on ordered pairs of process segments in the set of periodic processes P are satisfied.

Example 1.

Fig. 1 shows a feasible pre-run-time schedule S_O for all the segments in the primaries and alternates in the set of processes A, C, D, E, F, G, X on two processors that can be computed by the procedure in [15]. The following EXCLUSION relations on segments corresponding to critical sections are satisfied: $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $G_{P_{cs}}$, $G_{A_{cs}}$ and $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $X_{P_{cs}}$, $X_{A_{cs}}$. The EXCLUDES relations combined with the relative ordering of all the critical sections $A_{P_{cs}}$, $A_{A_{cs}}$, $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$, $X_{A_{cs}}$ define the following PREC relations: ($A_{P_{cs}}$ PREC $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$); ($G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$ PREC $A_{A_{cs}}$); ($A_{P_{cs}}$, $A_{A_{cs}}$ PREC $X_{A_{cs}}$).

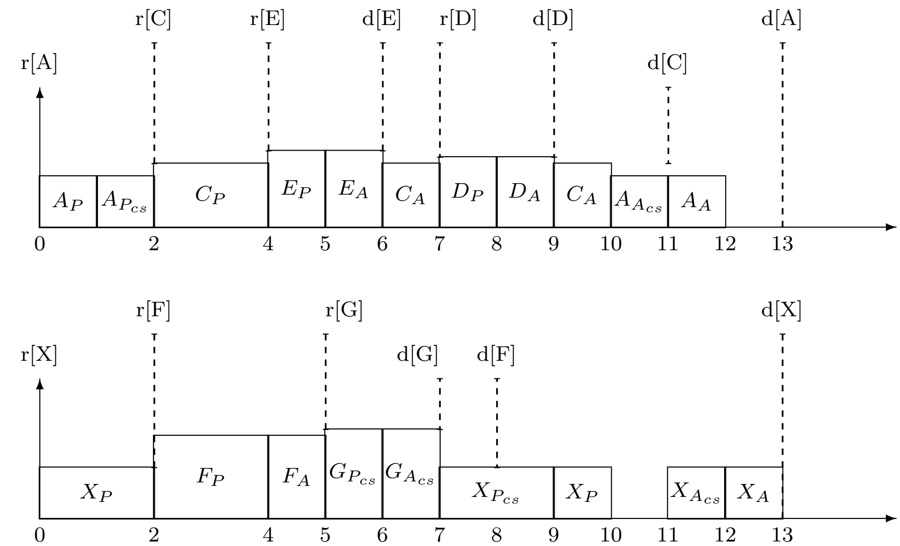

Fig. 1. Feasible pre-run-time schedule S_O for all the segments in the primaries and alternates in the set of processes A, C, D, E, F, G, X on two processors that can be computed by the procedure in [15].

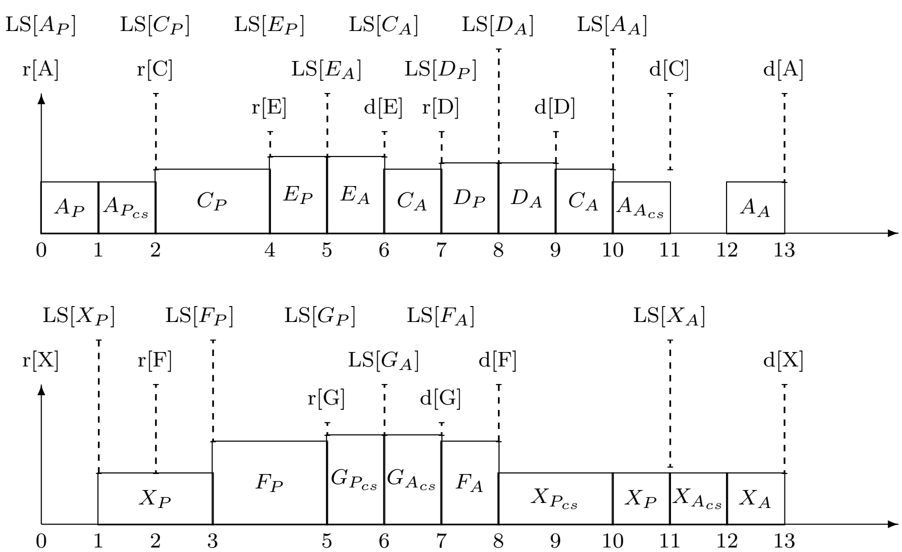

Fig. 2. Latest-start-time schedule S_L and the latest start times for all the primaries and alternates in the set of processes A, B, C, D, E, F, G, X, Y, Z on two processors that can be computed by the procedures described in [14] or [15] from the feasible pre-run-time schedule S_O in Fig. 1.

Example 2.

Fig. 2 shows a latest-start-time schedule S_L and the latest start times for all the primaries and alternates in the set of processes A, C, D, E, F, G, X on two processors that can be computed by the procedures described in [14] or [15] from the feasible pre-run-time schedule S_O in Fig. 1, such that the following EXCLUSION relations are satisfied: $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $G_{P_{cs}}$, $G_{A_{cs}}$ and $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $X_{P_{cs}}$, $X_{A_{cs}}$. The EXCLUDES relations combined with the relative ordering of all the critical sections $A_{P_{cs}}$, $A_{A_{cs}}$, $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$, $X_{A_{cs}}$ define the following PREC relations: ($A_{P_{cs}}$ PREC $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$); ($G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$ PREC $A_{A_{cs}}$); ($A_{P_{cs}}$, $A_{A_{cs}}$ PREC $X_{A_{cs}}$).

3 Run-Time Phase of the Method

3.1. Selecting Segments of Primaries and Alternates for Execution on a Multiprocessor At Run Time

At run-time, the segments of primaries and alternates are selected for execution on a multiprocessor at run-time according to the procedure described below:

Step (A)

At any time t, if the latest start time of any alternate p_A has been reached that is, $LS(p_A) = t$, then for each processor $m_1, \ldots, m_q, \ldots\ m_N$ in turn, select for execution on each processor m_q at time t a segment x of an alternate p_A that has the earliest deadline d[p] among all alternates for which the latest start time has been reached at time t, and which has not already been selected for execution on any processor at time t. If there exists some critical section segment x in p_A that was selected to execute on some processor m_q at time t, and there exists some uncompleted critical section segment y in primary p_{kP} or alternate p_{kA} such that y PREC x, then abort p_{kP} or p_{kA}. (This guarantees that all alternates will always be able to start on or before their respective latest start times and thus always be able to complete execution if they do not overrun.)

Step (B)

If after executing Step (A), there still exist some remaining processors that have not been assigned a process segment at time t, and if there exist any alternate p_A that has been activated that is, $ActivationTime(p_A) \leq t$, and alternate p_A has overrun and has not yet completed, then for each remaining processor m_q, select for execution on each processor m_q at time t a segment x of an alternate p_A that has the earliest deadline d[p] among all alternates A_p for which alternate p_A has been activated, that is, $ActivationTime(p_A) \leq t$, and alternate p_A has overrun and alternate p_A has not yet completed, and has not already been selected for execution on any processor at time t.

Step (C)

If after executing Step (B), there still exist some remaining processors that have not been assigned a process segment at time t, and if the latest start time of any primary p_P has been reached that is, $LS(p_P) = t$, then for each remaining processor m_q, select for execution on each processor m_q at time t a segment x of a primary p_P that has the earliest deadline d[p] among all primaries for which the latest start time has been reached at time t, and which has not already been selected for execution on any processor at time t, and such that the execution of segment x at time t will satisfy any PREC relation with any other segment y.

Step (D)

If after executing Step (C), there still exist some remaining processors that have not been assigned a process segment at time t, and if any alternate p_A has been activated that is, $ActivationTime(p_A) \leq t$, and has not completed, then for each remaining processor m_q, select for execution on each processor m_q at time t a segment x of an alternate p_A has been activated that is, $ActivationTime(p_A) \leq t$, and has not completed, and that has the earliest deadline d[p] among all alternates p_A that have been activated that is, $ActivationTime(p_A) \leq t$, and have not completed, and which have not already been selected for execution on any processor at time t, and such that the execution of segment x at time t will satisfy any PREC relation with any other segment y.

Step (E)

If after executing Step (D), there still exist some remaining processors that have not been assigned a process segment at time t, then for each remaining processor m_q, select for execution at time t a segment x of a primary p_P that has the earliest deadline d[p] among the set of all primaries that are ready and have not been selected for execution on any processor at time t, and such that the execution of segment x at time t will satisfy any PREC relation with any other segment y.

3.2. Main-Run-Time-Scheduler Method

At run-time, the main run-time scheduler uses the procedure described in Section 3.1 above for scheduling the segments, including critical sections, in primaries and alternates.

Given a latest-start-time schedule of all the primaries and alternates, at run-time there are the following main situations when the run-time scheduler may need to be invoked to perform a scheduling action:

(a) At a time t when some asynchronous process a has arrived and made a request for execution.

(b) At a time t when some segment of some primary p_P or alternate p_A or asynchronous process a has just completed its computation.

(c) At a time t that is equal to the latest start time $LS(p_P)$ of some primary p_P or the latest start time $LS(P_A)$ of some alternate p_A.

(d) At a time t that is equal to the release time R_{p_k} of some process p_k.

(e) At a time t that is equal to the deadline d_{p_i} of an uncompleted process p_i. (In this case, p_i has just missed its deadline, and the system should handle the error.)

(f) At a time t when some primary p_P generates a fault, in which case the corresponding alternate p_A will be activated, and the primary p_P will be aborted.

(g) At a time t when some alternate p_A generates a fault, and the system should handle the error.

In situation (a) above, the run-time scheduler is usually invoked by an interrupt handler responsible for servicing requests generated by an asynchronous process.

In situation (b) above, the run-time scheduler is usually invoked by a kernel function responsible for handling the completion of a segment of some primary p_P or alternate p_A or asynchronous process a.

In situations (c), (d), and (e) above, the run-time scheduler is invoked by programming the timer to interrupt at the appropriate time.

In situation (f) above, the run-time scheduler can be invoked by a hardware trap mechanism if a hardware fault in the primary p_P occurs, or by a software interrupt mechanism if a software fault in the primary p_P is detected.

In situation (g) above, an error handler is invoked by a hardware trap mechanism if a hardware fault in the alternate p_A occurs, or by a software interrupt mechanism if a software fault in the alternate p_A is detected.

Run-Time Scheduler Method

Let t be the current time.

Step 0. In situation (e) above, check whether any process p has missed its deadline d_p. If so perform error handling.

In situation (g) above, check whether any alternate p_A has generated a fault. If so perform error handling.

Step 1. In situation (a) above, if an A-h-k-a process a_i has arrived, execute the A-h-k-a Scheduler-Subroutine (the A-h-k-a Scheduler-Subroutine is described in [14]).

Step 2. In situation (f) above, if a primary p_P generates a fault, then the primary p_P will be aborted, and the corresponding alternate p_A will be activated; let $ActivationTime(p_A) = t$.

Step 3. Whenever the run-time scheduler is invoked due to any of the situations (b), (c) and (d) above at time t, do the following:

In situation (c) above, if the latest start time of an alternate p_A has been reached, that is, LS(p_A) $= t$, then the primary p_P will be aborted, and the corresponding alternate p_A will be activated; let $ActivationTime(p_A) = t$.

Recompute the latest start time $LS(p_P)$ or $LS(p_A)$ for each uncompleted primary p_P or alternate p_A that was previously executing at time $t - 1$ and has not overrun at time t using the procedures described in [14] or [15].

Any primary p_P or alternate p_A that was previously executing at time $t - 1$ but has either completed or has overrun at time t will be removed from the re-computed latest start time schedule.

Step 4. Use the method described in Section 3.1 to select up to N segments of primaries p_P or alternates p_A if possible to execute on the N processors at time t.

As mentioned in Section 3.1, If there exists some critical section segment x in p_A that was selected to execute on some processor m_q at time t, and there exists some uncompleted critical section segment y in primary p_{kP} or alternate p_{kA} such that y PREC x, then abort p_{kP} or p_{kA}. (This guarantees that all alternates will always be able to start on or before their respective latest start times and thus always be able to complete execution if they do not overrun, thus avoiding any cascading failures caused by primary or alternate critical section overruns.)

If any primary p_P has reached its latest start time LS(p_P) at time t, but was not selected to execute on any processor at time t, then abort primary p_P and activate its corresponding alternate p_A at time t; let $ActivationTime(p_A) = t$.

Step 5. At time 0 and after servicing each timer interrupt, and performing necessary error detection, error handling, latest start time re-calculations, and making scheduling decisions; - reset the timer to interrupt at the earliest time that any of the events (c), (d), and (e) above may occur.

Step 6. Let the segments of primaries p_P or alternates p_A that were selected in Step 4 start to execute at run-time t.

(If a selected segment belongs to a primary p_P or alternate p_A which was previously executing on some processor m_q at time $t - 1$, then one may let the selected segment in primary p_P or

alternate p_A continue to execute on the same processor m_q at time t.)
(End of Main Run-Time Scheduler)

It is noted here that the theoretical worst-case time complexity of all the steps in the Run-Time-Scheduler is $O(n)$.

Example 3.

Fig. 3 shows a possible run-time execution on two processors of all the segments in the primaries and alternates in the set of processes A, C, D, E, F, G, X shown in Fig. 1 of Example 1, in which the following EXCLUSION relations defined on segments that correspond to critical sections are satisfied: $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $G_{P_{cs}}$, $G_{A_{cs}}$ and $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $X_{P_{cs}}$, $X_{A_{cs}}$. The following PREC relations defined on the set of segments that correspond to critical sections are satisfied: ($A_{P_{cs}}$ PREC $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$); ($G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$ PREC $A_{A_{cs}}$); ($A_{P_{cs}}$, $A_{A_{cs}}$ PREC $X_{A_{cs}}$). In Fig. 3, C_P faults/underruns, while $X_{P_{cs}}$, F_A, $A_{A_{cs}}$, A_A overruns. The portions of the run-time execution during which $X_{P_{cs}}$, F_A, $A_{A_{cs}}$, A_A overruns are shown using dashed lines. In the pre-run-time phase, the procedures described in [14] or [15], will compute the latest start time values s of the primaries and alternates in the set of processes A, C, D, E, F, G, X shown in Fig. 2 in Example 2 for use at run time t = 0.

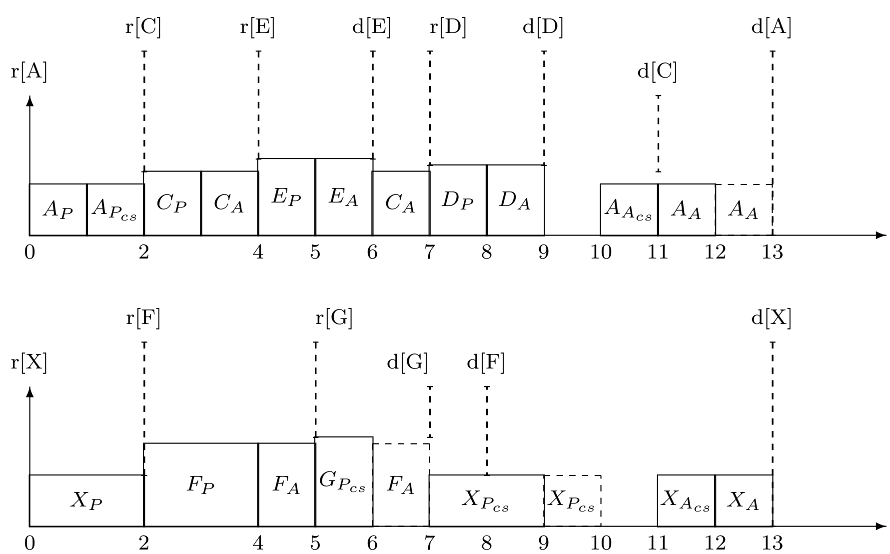

FIG. 3. Run-time schedule in which the following EXCLUSION relations are satisfied: $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $G_{P_{cs}}$, $G_{A_{cs}}$ and $A_{P_{cs}}$, $A_{A_{cs}}$ EXCLUDES $X_{P_{cs}}$, $X_{A_{cs}}$. The following PREC relations defined on the set of all the critical sections are satisfied: ($A_{P_{cs}}$ PREC $G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$); ($G_{P_{cs}}$, $G_{A_{cs}}$, $X_{P_{cs}}$ PREC $A_{A_{cs}}$); ($A_{P_{cs}}$, $A_{A_{cs}}$ PREC $X_{A_{cs}}$).

At run-time t = 0: the latest start time schedule is shown in Fig. 2. At t = 0, the latest start time of primary A_P is reached, so the run-time scheduler will select primary A_P in Step (C) to run on processor m_1. Then the run-time scheduler will select primary X_P in Step (E) to run on processor m_2, because X is the only other process that is ready at time t = 0. At t = 0, the timer will be programmed to interrupt at C_P's latest start time LS(C_P) = 2, before actually dispatching A_P and X_P for execution.

At time t = 2: the timer interrupts at C_P's latest start time LS(C_P) = 2; while $A_{P_{cs}}$ generates a fault, which causes the primary segment $A_{P_{cs}}$ to be aborted and the alternate segment $A_{A_{cs}}$ to be activated. After re-computing the latest-start-times, LS(X_P) = 8. The run-time scheduler will first select primary C_P in Step (C) to run on processor m_1, because primary C_P's deadline $d[C] = 11$ is the earliest deadline among all primaries for which the latest start time has been reached. Then the run-time scheduler will select primary F_P to run on processor m_2 in Step (E), because there are no remaining alternates that have been activated or primaries for which the latest start time has been reached, and F_P has the earliest deadline among all remaining primaries that are ready, d(F) = 8.
At t = 2, the timer will be programmed to interrupt at primary F_P's latest-start-time LS(F_P) = 3, before actually dispatching C_P and F_P for execution.

At time t = 3: primary C_P is aborted after C_P generates a fault, causing alternate C_A to be activated. After re-computing the latest-start-times for F_P at time 3, LS(F_P) = 4. The run-time scheduler will first select alternate C_A to run on processor m_1 in Step (D), because alternate C_A's deadline $d[C] = 12$ is the earliest deadline among all alternates that have been activated. Then the run-time scheduler will select primary F_P to run on processor m_2 in Step (E), because there are no remaining alternates that have been activated or primaries for which the latest start time has been reached, and F_P has the earliest deadline among all remaining primaries that are ready, d(F_P) = 8.
At t = 3, the timer will be programmed to interrupt at primary E_P's latest start time LS(E_P) = 4, before actually dispatching C_A and F_P for execution.

At time t = 4: primary F_P is aborted after F_P generates a fault, causing alternate F_A to be activated. At t = 4, the latest start time of primary E_P, LS(E_P) = 4 is also reached. After re-computing the latest-start-times, LS(C_A) = 9.
The run-time scheduler will select primary E_P in Step (C) and select alternate F_A in Step (D) to run on processor m_1 and processor m_2 respectively, because E_P has the earliest deadlines d[E] = 6 among all primaries for which the latest start time has been reached; and F_A has the earliest deadlines $d[F] = 8$ among all alternates that have been activated.
At t = 4, the timer will be programmed to interrupt at alternate E_A's latest start time LS(E_A) = 5, which is equal to primary G_P's latest start time LS(G_P) = 5, before actually dispatching F_A and E_P for execution.

At time t = 5: alternate E_A's earliest start time LS(E_A) = 5 has been reached, hence alternate E_A is activated while primary E_P is cancelled. The run-time scheduler will select alternate E_A in Step (A) to run on processor m_1. At t = 5 the latest start time of primary $G_{P_{cs}}$ has been reached, so the run-time scheduler will select primary $G_{P_{cs}}$ in Step (C) to run on processor m_2.

At time t = 6: E_A and G_P both complete. F_A overruns at t = 6, so the run-time scheduler will select the overrunning alternate F_A in Step (B) to run on processor m_2. The run-time scheduler will select alternate C_A in Step (D) to run on processor m_1.

At time t = 7: primary D_P's latest start time has been reached, so the run-time scheduler will select primary D_P in Step (C) to run on processor m_1. F_A completes its execution after overrunning. C_A completes its execution. The run-time scheduler will select primary critical section $X_{P_{cs}}$ in Step (E) to run on processor m_2.

At time t = 8: alternate D_A's latest start time has been reached, so alternate D_A is activated while primary D_P is cancelled. The run-time scheduler will select alternate D_A in Step (A) to run on processor m_1. The run-time scheduler will select primary critical section $X_{P_{cs}}$ in Step (E) to run on processor m_2.

At time t = 9: alternate D_A completes while the critical section $X_{P_{cs}}$ in primary X_P starts to overrun. The run-time scheduler selects the critical section $X_{P_{cs}}$ in primary X_P in Step (E) to execute on processor m_2. Because critical section $X_{P_{cs}}$ in primary X_P has not completed its execution at time t = 9, the run-time scheduler cannot select the critical section $A_{A_{cs}}$ in alternate A_A to run on processor $m1$ in Step (D) because of the PREC relation ($X_{P_{cs}}$ PREC $A_{A_{cs}}$). This causes processor m_1 to become idle from time t = 9 to time t = 10.

At time t = 10: the latest start time of critical section $A_{A_{cs}}$ in alternate A_A has been reached, but critical section $X_{P_{cs}}$ in primary X_P has not yet completed. The run-time scheduler will select critical section $A_{A_{cs}}$ in alternate A_A in Step (A) to run on processor m_1. Note that the software architecture guarantees that all alternates will be able start execution on or before their respective latest start times and complete execution as long as they do not overrun. Due to the PREC relation ($X_{P_{cs}}$ PREC $A_{A_{cs}}$), primary X_P will be aborted at t = 10 at the end of Step (A) in the method for selecting segments of primaries and alternates on a multiprocessor at run-time in Section 3.1. After primary X_P is aborted, alternate critical section $X_{A_{cs}}$ will be activated at time t = 10. But the run-time scheduler cannot select alternate $X_{A_{cs}}$ to run at time t = 10 in Step (E) because there exists the PREC relation ($A_{A_{cs}}$ PREC $X_{A_{cs}}$) and $X_{A_{cs}}$ has not yet completed execution at time t = 10. Since no other process can be scheduled to execute on processor m_2 at t = 10, processor m_2 is idle from t = 10 to 11.

At time t = 11: the alternate critical section $A_{A_{cs}}$ in alternate A_A completes its execution at time t = 11. The run-time scheduler will select alternate $X_{A_{cs}}$ in Step (A) to run on processor m_2 while satisfying the PREC relation ($X_{A_{cs}}$ PREC $X_{A_{cs}}$). The run-time scheduler will select alternate A_A in Step (D) to run on processor m_1.

At time t = 12: the alternate critical section $X_{A_{cs}}$ completes while alternate A_A overruns. The run-time scheduler will select alternate X_A in Step (A) to run on processor m_2. The run-time scheduler will select the overrunning alternate A_A in Step (B) to run on processor m_1.

At time t = 13: alternate A_A completes before its deadline despite overrunning. Alternate X_A also completes before its deadline.

4 Conclusions

We present a software architecture for handling complex critical section constraints on multiprocessors in a real-time embedded system which uses a primary and an alternate for each real-time task to achieve fault tolerance. The software architecture allows both primaries and alternates to have critical sections in which shared data structures can be read and updated while keeping the shared data in a consistent state. The software architecture also allows both primaries and alternates to either overrun or underrun, while still guaranteeing that all primaries or alternates that do not overrun will always meet their deadlines . Thus the software architecture significantly increases system robustness and reliability on a multiprocessor in a fault tolerant real-time embedded system.

References

[1] Laprie, J.C., 1985, "Dependable computing and fault tolerance: concepts and terminology." *Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15)*, pp. 2-11, 1985.

[2] Avizienis, A., Laprie, J.C. Randell, B., and Landwehr C., 2004, "Basic concepts and taxonomy of dependable and secure Computing." *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 1, 2004.

[3] Han, C-C., Shin, K.G., and Wu, J., 2003, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults." *IEEE Trans. on Computers*, Vol. 52, No. 3, March 2003.

[4] Lima, G.M.D., and Burns, A., 2003, "An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems." *IEEE Trans. on Computers*, Vol. 52, No. 10, October 2003.

[5] Manimaran G., and Murphy, C.S.R., 1998, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. " *IEEE Trans. Parallel and Distr. Sys.*, vol. 9, no. 11, Nov. 1998.

[6] Liestman A.L., and Campbell, R.H, 1986, "A fault-tolerant scheduling problem. " *IEEE Trans. Software Eng.*, vol. 12, no. 11, Nov. 1986.

[7] Chetto, H.,.and Chetto, M., 1989, "Some Results of the earliest deadline scheduling algorithm." *IEEE Trans. Software Eng.*, vol. 15, no. 10, pp. 1261-1269, Oct. 1989.

[8] Koren, G., and Shasha, D., 1995, "Dover: an optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems." *SIAM Journal on Computing*, Vol. 24, no. 2, pp. 318-339.

[9] Gardner, M. K., and Liu, J. W. S., 1999, "Performance of algorithms for scheduling real-time systems with overrun and overload," *Proc. 11th Euromicro Conference on Real-Time Systems*, England, pp. 9-11.

[10] Stewart, D. B., and Khosla, 1997, "Mechanisms for detecting and handling timing errors," *Communications of the ACM*, vol. 40, no. 1, pp. 87-90.

[11] Xu, J., 1993, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engineering*, Vol. 19 (2), pp. 139-154.

[12] Xu, J. and Parnas, D. L., 1990, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engineering*, Vol. 16 (3), pp. 360-369. Reprinted in *Advances in Real-Time Systems*, edited by Stankovic, J. A. and Ramamrithan, K., IEEE Computer Society Press, 1993, pp. 140-149.

[13] Caccamo, M., Buttazzo, G. C., and Thomas, D. C., 2005, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Tran. Computers*, vol. 54, n. 2, pp. 198-213.

[14] Xu, J., 2017, "Efficiently handling process overruns and underruns on multiprocessors in real-time embedded systems," *13th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, Cleveland, Ohio, USA, on August 6-9, 2017.

[15] Xu, J., 2018, "Handling process overruns and underruns on multiprocessors in a fault-tolerant real-time embedded system," *14th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, Oulu, Finland, on July 1-4, 2018.

[16] Haerder, T. and Reuter, A., (1983), "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol 15, n. 4, pp. 287.