Handling Process Overruns and Underruns on Multiprocessors in a Fault-Tolerant Real-Time Embedded System

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

***Abstract*—The failure of safety-critical hard real-time embedded systems, can have catastrophic consequences. In such systems, a *fault tolerant design* is often necessary to enable the system to continue to provide a specified service, possibly at a reduced level of performance, rather than failing completely, in spite of system errors. 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. This paper presents a method which provides a higher level of system dependency and reliability by effectively handling underruns and overruns 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 main advantage of this method is that it significantly increases the chances that either the primary or the alternate of each process will be able to successfully complete its computation before its deadline despite overrunning, which significantly increases system robustness and reliability, while at the same time any additional processor capacity created at run-time due to primary or alternate underruns can be efficiently utilized, which increases system resource and processor utilization, while also satisfying additional complex constraints defined on the primaries and alternates such as precedence and exclusion relations.**

I. INTRODUCTION

The failure of safety-critical hard real-time embedded systems, such as commercial aircraft control systems, nuclear reactor control systems, industrial process control systems, can have catastrophic consequences. In such systems, a *fault tolerant design* is often necessary to enable the 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. In general, fault tolerance requires *redundancy*, such as the ability to perform multiple computations, in which a same function is implemented via separate designs and implementations, i.e., through *design diversity* [1][2]. 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 this paper a method is presented which provides a higher level of system dependency and reliability by effectively handling underruns and overruns in a fault tolerant real-time embedded system which uses the primary and alternate approach for achieving fault tolerance.

It is possible to provide a higher level of system dependency and reliability if one can effectively handle underruns and overruns of both primaries and alternates. This is because in order to decrease average-case response times, most modern computer architectures commonly employ a variety of nondeterministic technologies such as interrupts, DMA, pipelining, caching, prefetching, etc. Unfortunately, it can be extremely difficult to accurately estimate the worst-case computation times of real-time primaries and alternates due to the widespread use of such technologies [10][13]. This can have very undesirable consequences: overestimating worst-case computation times will cause primaries and alternates to underrun and result in low processor utilization; but underestimating worst-case computation times can cause primaries and alternates to overrun and cause real-time tasks to miss deadlines, which may cause the whole system to crash.

With the method presented in this paper a *"latest start time "* LS(p_P) is computed for every uncompleted primary p_P and a *"latest start time"* LS(p_A) is computed for every alternate p_A that has not overrun for each process p on a multiprocessor, which satisfy the following properties:
(1) if every primary p_P starts execution on or before its respective latest start time LS(p_P), and does not fault or overrun, then every primary p_P is guaranteed to complete its computation on or before its process deadline d(p);
(2) even if every primary p_P, after starting execution at its respective latest start time LS(p_P), *generates an error, aborts, and activates its corresponding alternate p_A precisely at the end of every primary p_P's worst-case computation time c(p_P)*, every alternate p_A is still guaranteed to be able to:
(2a) start execution on or before its respective latest start time LS(p_A); and
(2b) complete its computation on or before its process deadline d(p) as long as p_A does not fault or overrun;
(3) if any primary overruns, that is, does not complete after executing for a number of time units equal to its worst-case computation time, then that primary can continue to execute at any time t on any processor m, as long as no other primary or alternate with an earlier deadline is prevented from starting on or before its latest start time, while guaranteeing that every other primary p_P and every other alternate p_A will still be able to:
(3a) start execution on or before its respective latest start time LS(p_P) or LS(p_A);

978-1-5386-4643-4/18/$31.00 ©2018 IEEE

(3b) complete its computation on or before its process deadline d(p) as long as it does not fault or overrun.
Thus this method is able to efficiently utilize any spare capacity in the system, including any spare capacity created at run-time due to primary or alternate normal underruns or primary underruns due to errors. A primary advantage of this method is that it significantly increases the chances that either the primary or the alternate of each process will be able to successfully complete its computation before its deadline despite overrunning, which significantly increases system robustness and reliability, while at the same time any additional processor capacity created at run-time due to primary or alternate underruns can be efficiently utilized, which increases system resource and processor utilization. This method also has many other very important advantages, such as being able to handle more complex constraints and avoid the use of complex synchronization mechanisms.

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]. A significant contribution of the work presented in this paper, is that, to our knowledge, this is the first time that a method has been devised that is capable of computing a "latest start time" on a multiprocessor for every uncompleted primary or alternate in a real-time system, and use the latest start times, to effectively handle underruns and overruns for both primaries and alternates on a multi-processor, such that the three properties described above are satisfied, while also satisfying additional complex constraints defined on the primaries and alternates such as precedence and exclusion relations. The method significantly increases the chances that either the primary or the alternate of each process 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.

II. REAL-TIME PERIODIC PROCESSES, PRIMARIES, ALTERNATES

A periodic process p can be described by 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 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*.
The worst case computation time c_p required by process p is equal to the *sum* of the *worst case computation time* c_{p_P} *required by the primary* p_P and the *worst case computation time* c_{p_A} *required by the alternate* p_A, that is, $c_p = c_{p_P} + c_{p_A}$.

If a process p *PRECEDES* process p_1, then primary p_{1P} or alternate p_{1A} cannot start execution before either primary p_P or alternate p_P has completed its computation.
If a process p *EXCLUDES* process p_1, then neither primary p_P or alternate p_A is allowed to preempt primary p_{1P} or alternate p_{1A} at any time. Exclusion relations can be used to prevent concurrent processes from simultaneously accessing shared resources such as shared memory [11][12][14][15].

III. METHOD FOR COMPUTING A FEASIBLE PRE-RUN-TIME SCHEDULE FOR PRIMARIES AND ALTERNATES OF PROCESSES

Notation
s(x): $s(x)$ is the time of the beginning of the time slot that was reserved for either a process, or a primary, or an alternate x in the pre-run-time schedule.
s'(x): $s'(x)$ is the actual time that either a process, or a primary, or an alternate x was/will be put into execution at run-time. At any time t, if either a process, or a primary, or an alternate x has been put into execution after or at time t, then $t \leq s'(x)$ is true, otherwise $\neg(t \leq s'(x))$ is true.
e(x): $e(x)$ is the time of the end of the time slot that was reserved for either a process, or a primary, or an alternate x in the pre-run-time schedule.
e'(x): $e'(x)$ is the actual time at which either a process, or a primary, or an alternate x's execution ends at run-time. At any time t, if either a process, or a primary, or an alternate x's execution has ended after or at time t, then $t \leq e'(x)$ is true, otherwise if x's execution has not ended before or at time t, then $\neg(t \leq e'(x))$ is true.
C'(x): $C'(x)$ is the remaining worst-case computation time of either a process, or a primary, or an alternate x at run-time.

The procedure below computes a *"feasible-pre-run-time schedule"* S_O on a multiprocessor, for a set of uncompleted periodic processes P, in which arbitrary PRECEDES and EXCLUDES relations defined on ordered pairs of processes in P are satisfied. This will also produce a feasible pre-run-time schedule for all the primaries and alternates in those processes.

Let the set of processors be $M = \{ m_1, \ldots, m_q, \ldots, m_N \}$. In the procedure below, "p_j on m_q" means "process p_j has been previously assigned processor time on processor m_q". "$s(p_j)$" refers to the "start time" of p_j, or the beginning (left hand side) of p_j's time slot in the pre-run-time schedule S_O.

```
t ← 0
while ¬(∀p_i ∈ P : e(p_i) ≤ t) do
  begin
    for m_q = m_1 to m_N do
      begin
        Among the set
        { p_j ∈ P | ((¬(s(p_j) ≤ t) ∨ (p_j on m_q ∧ (s(p_j) < t))
        % p_j not started yet or p_j started on m_q
        ∧(r'(p_j) ≥ t) ∧ ¬(e(p_j) ≤ t)
```

% p_j ready and p_j uncompleted
$\wedge(\nexists p_k \in P : (p_k\ EXCLUDES\ p_j)$
$\wedge(s(p_k) < t) \wedge \neg(e(p_k) \leq t))$
% no p_k that has started but not completed
% such that p_k EXCLUDES p_j
$\wedge(\nexists p_k \in P : (p_k\ PRECEDES\ p_j) \wedge \neg(e(p_k) \leq t))$
% no uncompleted p_k such that p_k PRECEDES p_j
}
select the process p_j that has min dp_j.
in case of ties, select the process p_j that has a smaller process index number j.
if $\neg(s(p_j) \leq t)$ then $s(p_j) \leftarrow t$
assign the time unit [t, t + 1] on m_q to p_j's time slot in the pre-run-time schedule S_O.
if the total number of time units assigned to p_j's time slot is equal to c_{p_j}, then $e(p_j) \leftarrow t$
end
$t \leftarrow t\ +\ 1$
end

Once a feasible pre-run-time schedule has been computed for the given set of processes which satisfies all the given constraints, one can easily obtain the start time s(p_P) and end time e(p_P) for each primary p_P, and the start time s(p_A) and end time e(p_A) for each alternate p_A that corresponds to each process p as follows. The start time s(p_P) for each primary p_P is equal to the start time s(p) for process p in the feasible pre-run-time schedule computed by the procedure described above. If the total length of the portion of the time slot allocated to the primary p_P is x, then the end time e(p_P) for each primary p_P is equal to the end of the xth time unit of the time slot allocated to process p; the start time s(p_A) for each alternate p_A is equal to the beginning of the $x+1$th time unit in the time slot allocated to the process p; the end time e(p_A) for each alternate p_A is equal to the end time e(p) for process p in the latest start time schedule computed by the procedure described above.

Given any original feasible pre-run-time schedule S_O on a multiprocessor, a set of "PREC" relations on ordered pairs of processes in the set of periodic processes P in the original pre-run-time schedule S_O is defined as follows:
$\forall p_i \in P, p_j \in P,$
if $e(p_i) < e(p_j) \wedge$ ((p_i EXCLUDES p_j) $\vee$ (p_i PRECEDES p_j))
then let p_i PREC p_j

If a "PREC" relation between two processes, p_i PREC p_j holds, then the "PREC" relation will also hold between the primaries and alternates of the two processes, that is: p_{iP} (p_{iA}) PREC p_{jP} (p_{jA}).

Example 1.

Fig. 1 shows a feasible pre-run-time schedule S_O 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 procedure above. The following EXCLUSION and PRECEDES relations are satisfied: D EXCLUDES Y and D PRECEDES Y (D PREC Y).

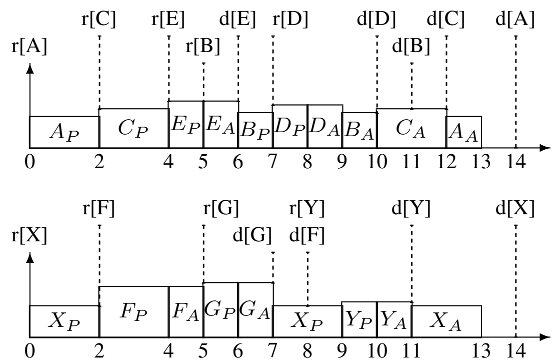

Fig. 1. Feasible pre-run-time schedule S_O for all the primaries and alternates in the set of processes A, B, C, D, E, F, G, X, Y, Z on two processors that can be computed by the procedure above. The following EXCLUSION and PRECEDES relations are satisfied: D EXCLUDES Y and D PRECEDES Y (D PREC Y).

IV. METHOD FOR COMPUTING LATEST START TIMES FOR PRIMARIES AND ALTERNATES

Given an original 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 processes 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 are satisfied.

Once a latest-start-time schedule for the given set of processes which satisfies all the given constraints has been computed, the latest start time LS(p_P) for each primary p_P, and the latest start time LS(p_A) for each alternate p_A that corresponds to each process p can be obtained as follows. The latest start time LS(p_P) for each primary p_P is equal to the latest start time LS(p) for process p, which is the start time of the time slot allocated to process p in the latest-start-time schedule computed by the procedures described in [14] or [15]. If the total length of the portion of the time slot allocated to the primary p_P is x, then the latest start time LS(p_A) for each alternate p_A is equal to the beginning of the $x+1$th time unit in the time slot allocated to the entire process p in the latest start time schedule computed by the procedures described in [14] or [15].

Example 2.

Fig. 2 below 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, 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, such that the following EXCLUSION and PRECEDES relations are satisfied: D EXCLUDES Y and D PRECEDES Y (D PREC Y).

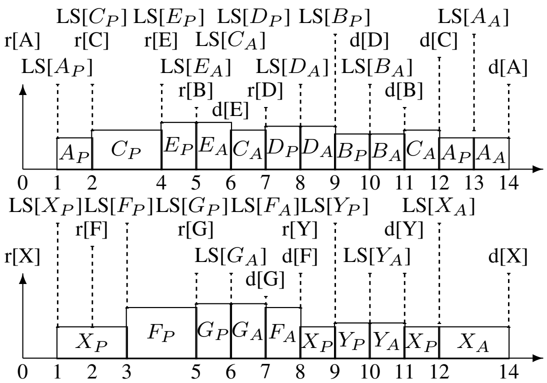

Fig. 2. Latest-start-time schedule S_L and the latest start times for all the primaries and alternates in the set of processes A, B, C, D, E, F, G, X, Y, Z on two processors that can be computed by the procedures described in [14] or [15] from the feasible pre-run-time schedule S_O in Fig. 1. The following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

V. RUN-TIME PHASE OF THE METHOD

5.1. Method For Selecting Primaries and Alternates for Execution on a Multiprocessor At Run Time

It is noted here that any method for selecting primaries and alternates for execution on a multiprocessor at run-time should attempt to achieve an appropriate balance in the trade off between the following two considerations:

Consideration 1. For any real-time process p, completing the primary p_P is the *first choice*, that is, completing the primary p_P is preferable to completing the alternate p_A. It is for this reason that an alternate p_A will not be activated unless an error in the primary p_P is detected, or the successful completion of the primary p_P cannot be guaranteed. It is also for this reason that the following method allows any primary p_P to continue to overrun as long as the overrunning primary p_P does not prevent any other primary p_{1P} or alternate p_{1A} from starting at their latest start times LS(p_{1P}) or LS(p_{1A}).

Consideration 2. However, once a primary p_P has been aborted and an alternate p_A has been activated, then successful completion of that alternate p_A before its deadline $d(p)$ is considered to be the *"last chance to avoid failure of the task/process "* p, and potentially, the *"last chance to avoid failure of the entire system."* It is for this reason that the following method will give any alternate p_A that has been activated with an earlier deadline higher priority than any other primary p_{1P} or alternate p_{1A} with a later deadline ($d(p) < d(p_1)$), when selecting primaries or alternates for execution.

Step (A)

At any time t, if the latest start time of any alternate p_A has been reached that is, $LS(p_A) = t$, or if the latest start time of any primary p_P has been reached that is, $LS(p_P) = t$, or if any alternate p_A has been activated that is, $ActivationTime(p_A) \leq t$, then for each processor m_1, ..., m_q, ... m_N in turn, select for execution on each processor m_q at time t an alternate p_A or a primary p_P that has the earliest deadline d[p] among all alternates that have been activated or 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.

At any time t
 if ($\exists p \in P$: (($LS(p_A) = t \vee LS(p_P) = t$)
 $\vee$ (($ActivationTime(p_A) \leq t$) $\wedge \neg(e'(p) \leq t$))) then
 for m_q = m_1 to m_N do
 begin
 Among the set
 { $p \in P$ | ((p_P or p_A has not been selected for execution at time t)
 $\wedge$ (($LS(p_A) = t \vee LS(p_P) = t$)
 $\vee$ (($ActivationTime(p_A) \leq t$) $\wedge \neg(e'(p) \leq t$)))
 $\wedge(\nexists p_k \in P : p_k\ PREC\ p \wedge \neg(e'(p_k) \leq t)))$
 % no uncompleted p_k such that p_k PREC p
 }
 select the alternate p_A or primary p_P that has min $d(p)$.
 in case of ties, select an alternate p_A over any primary p_P.
 assign the selected p_A or p_P to execute on m_q at time t.
 end

Example 3. At time t = 4 in the run-time execution scenario illustrated in Fig. 3, primary F_P is aborted after F_P generates a fault, causing alternate F_A to be activated, $ActivationTime(F_A)$ = 4. 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, the latest start times are shown in Fig. 6. The run-time scheduler will select primary E_P and alternate F_A to run on processor m_1 and processor m_2 respectively in Step (A), because E_P and F_A have the earliest deadlines d[E] = 6 and $d[F]$ = 8 among all alternates that have been activated or primaries for which the latest start time has been reached.

Step (B)

If after executing Step (A), there still exist some remaining processors that have not been assigned a process at time t, then for each remaining processor m_q, select for execution at time t 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.

for each remaining unassigned processor m_q do
 begin
 Among the set
 { $p \in P$ | ((p_P has not been selected for execution at time t)
 $\wedge$ ($r'[p] \leq t$)
 $\wedge(\nexists p_k \in P : p_k\ PREC\ p \wedge \neg(e'(p_k) \leq t)))$
 % no uncompleted p_k such that p_k PREC p
 }
 select the primary p_P that has min $d(p)$.
 in case of ties, select a primary p_P that has the earliest latest start time LS(p_P).
 assign the selected p_P to execute on m_q at time t.
 end

Example 4. At time t = 2 in the run-time execution scenario illustrated in Fig. 3, C_P's latest start time LS(C_P) = 2 has been reached; while A_P overruns. After re-computing the latest-start-times, the latest start times are shown in Fig. 4. In Step (A), run-time scheduler will first select primary C_P to run on processor m_1, because primary C_P's deadline $d[C]$ = 12 is the earliest deadline among all alternates that have been activated or primaries for which the latest start time has been reached. Then in Step (B), the run-time scheduler will select primary F_P to run on processor m_2, 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.

Discussion: It is also possible to employ an alternative method for selecting primaries and alternates for execution on a multiprocessor at run time, in which *any alternate* p_A *that has been activated is assigned a higher priority than any primary* p_{1P}*, even if the primary* p_{1P} *has an earlier deadline and the latest start time of primary* p_{1P} *has been reached, that is,* $d(p_1) < d(p)$ *and* $LS(p_{1P}) = t$. Note that such an alternative method prioritizes Consideration 2 over Consideration 1; it has the advantage of further increasing the probability that an activated alternate p_A will be able to successfully complete before its deadline $d(p)$, but has the disadvantage of decreasing the probability that "first choice" primaries will be able to successfully complete. Another possible way to fine-tune the balance in making such tradeoffs would be to assign different relative priorities to different alternate-primary pairs based on the specific characteristics of each alternate and primary in the real-time embedded system application.

5.2. Main-Run-Time-Scheduler Method

At run-time, the main run-time scheduler uses the methods described in Section 4.1 above for scheduling primaries and alternates.

With this method, 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 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 primary p_P or alternate p_A or asynchronous process a completions.

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.

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 4.1 to select up to N primaries p_P or alternates p_A if possible to execute on the N processors at time t.

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 primaries p_P or alternates p_A that were selected in Step 4 start to execute at run-time t.
(If a selected primary p_P or alternate p_A was previously executing on some processor m_q at time $t-1$, then one may let selected 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 $O(n)$. The method in this paper is also applicable on single core architectures.

Example 5.

Fig. 3 shows a possible run-time execution on two processors of the primaries and alternates in the set of processes A, B, C, D, E, F, G, X, Y, Z shown in Fig. 1 of Example 1. In Fig. 3, C_P faults/underruns, while C_A, F_A, A_P, X_A overruns. The portions of the run-time execution during which C_A, F_A, A_P, X_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, B, C, D, E, F, G, X, Y, Z shown in Fig. 2 in Example 2 for use at run time t = 0.

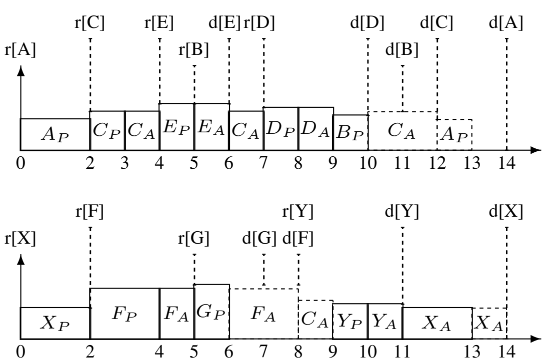

FIG. 3. Run-time schedule (D PREC Y).

At run-time t = 0: the latest start time schedule is shown in Fig. 2. The run-time scheduler will select primary A_P and primary X_P to run on processor m_1 and processor m_2 respectively in Step (B), because A and X are the only processes that are 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 overruns. After re-computing the latest-start-times, LS(X_P) = 8, as shown in Fig. 4. The run-time scheduler will first select primary C_P to run on processor m_1 in Step (A), because primary C_P's deadline $d[C]$ = 12 is the earliest deadline among all alternates that have been activated or 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 (B), 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 E_P's latest-start-time LS(E_P) = 4, 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, as shown in Fig. 5. The run-time scheduler will first select alternate C_A to run on processor m_1 in Step (A), because alternate C_A's deadline $d[C]$ = 12 is the earliest deadline among all alternates that have been activated or 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 (B), 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) = 11 as shown in Fig. 6.
The run-time scheduler will select primary E_P and alternate F_A to run on processor m_1 and processor m_2 respectively in Step (A), because E_P and F_A have the earliest deadlines d[E] = 6 and $d[F]$ = 8 among all alternates that have been activated or primaries for which the latest start time has been reached.
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. F_A overruns. The latest-start-times are shown in Fig. 7.

At time t = 6: E_A and G_P both complete. The latest-start-times are shown in Fig. 8.

At time t = 7: primary D_P's latest start time has been reached. F_A and C_A still overrun. The latest-start-times at time 7 are shown in Fig. 9.

At time t = 8: alternate F_A completes; alternate D_A's latest start time has been reached, so alternate D_A is activated while primary D_P is cancelled. F_A and C_A still overrun. The latest-start-times at time 8 are shown in Fig. 10.

At time t = 9: alternate D_A completes; B_P and Y_P's latest start times have been reached. C_A still overruns. The latest-start-times at time 9 are shown in Fig. 11.

At time t = 10: primary B_P completes; alternate Y_A's latest start time has been reached, so alternate Y_A is activated while primary Y_P is cancelled. The latest-start-times at time 8 are shown in Fig. 12.

At time t = 11: Y_A completes while C_A still overruns. The latest-start-times at time 11 are shown in Fig. 13.

At time t = 12: C_A completes before its deadline despite overrunning. The latest start times are shown in Fig. 14.

At time t = 13: primary A_P completes before its corresponding alternate A_A latest start time LS(A_A) = 13 after overrunning. Alternate X_A still overruns.

At time t = 14: alternate X_A completes before its deadline despite overrunning.

VI. CONCLUSIONS

This paper presents a method which provides a higher level of system dependency and reliability by effectively handling underruns and overruns in a fault tolerant real-time embedded system which uses a *primary* program and an *alternate* program for each real-time task to achieve fault tolerance. A main advantage of this method is that it significantly increases the chances that either the primary or the alternate of each process will be able successfully complete its computation before its deadline despite overrunning, which significantly increases system robustness and reliability, while at the same time any additional processor capacity created at run-time due to primary or alternate underruns can be efficiently utilized, which increases system resource and processor utilization, while also satisfying additional complex constraints defined on the primaries and alternates such as precedence and exclusion relations.

VII. APPENDIX: RECOMPUTED LATEST START TIME SCHEDULES IN EXAMPLE 5

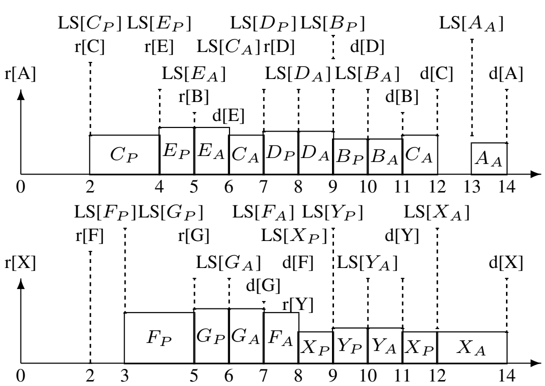

FIG. 4. Latest start times at run-time t = 2 (D PREC Y).

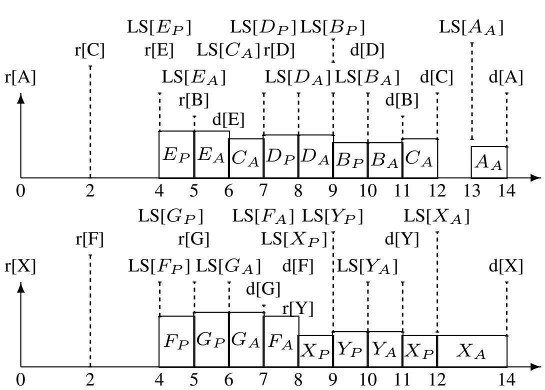

FIG. 5. Latest start times at run-time t = 3 (D PREC Y).

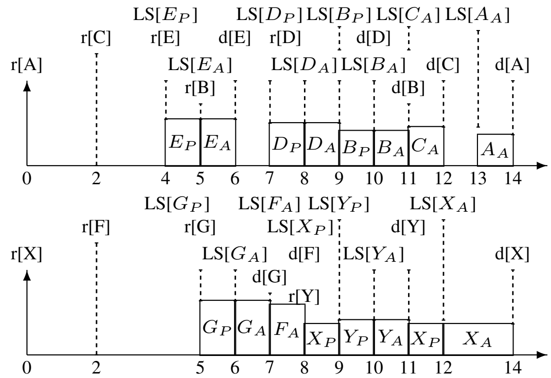

FIG. 6. Latest start times at run-time t = 4 (D PREC Y).

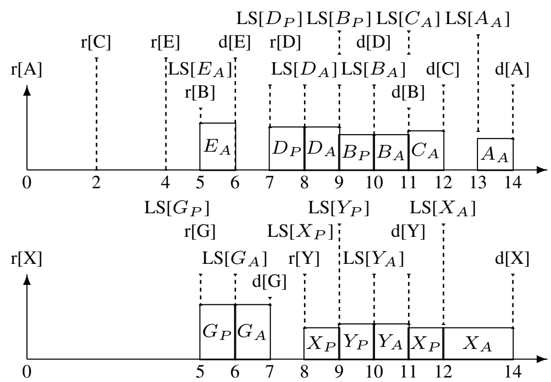

FIG. 7. Latest start times at run-time t = 5 (D PREC Y).

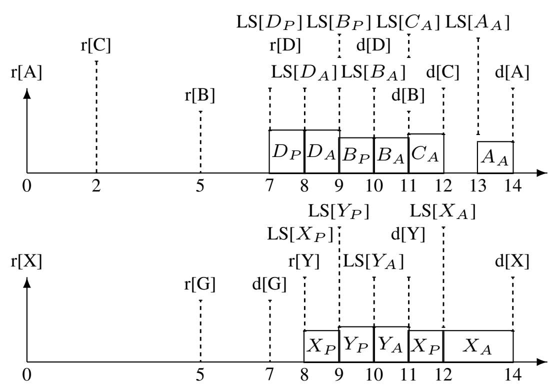

FIG. 8. Latest start times at run-time t = 6 (D PREC Y).

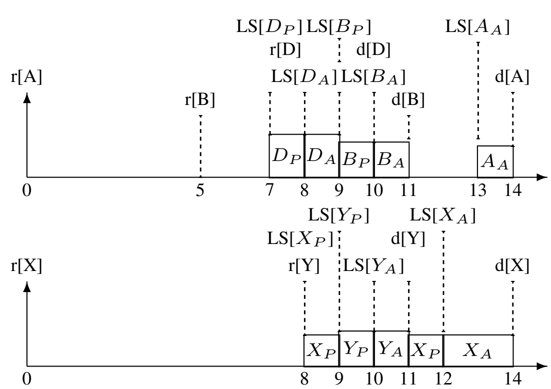

FIG. 9. Latest start times at run-time t = 7 (D PREC Y).

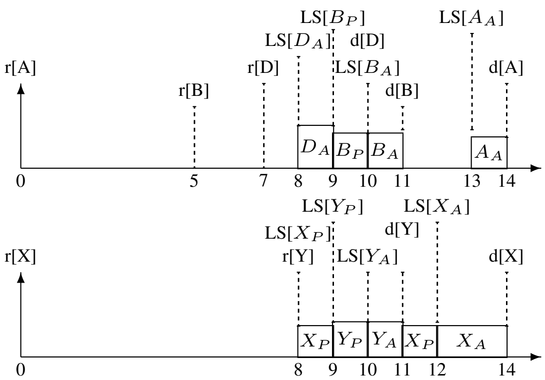

FIG. 10. Latest start times at run-time t = 8 (D PREC Y).

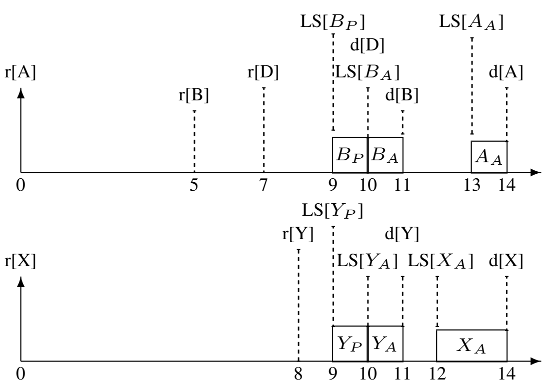

FIG. 11. Latest start times at run-time t = 9.

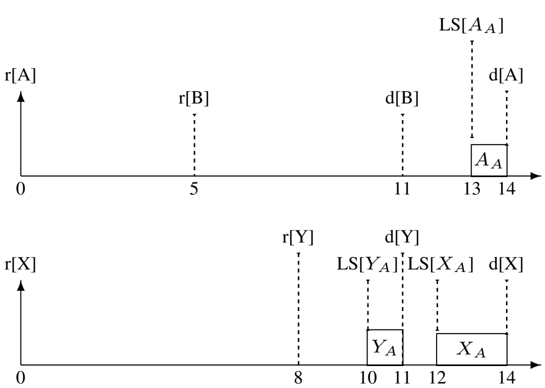

FIG. 12. Latest start times at run-time t = 10.

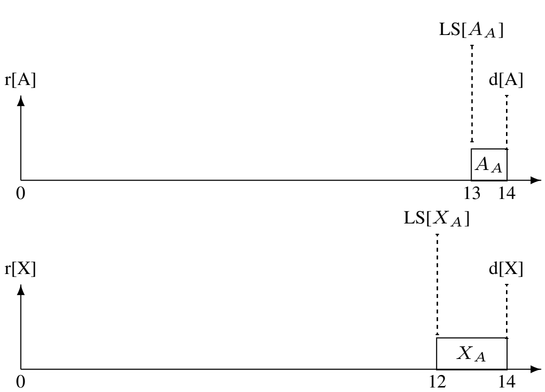

FIG. 13. Latest start times at run-time t = 11.

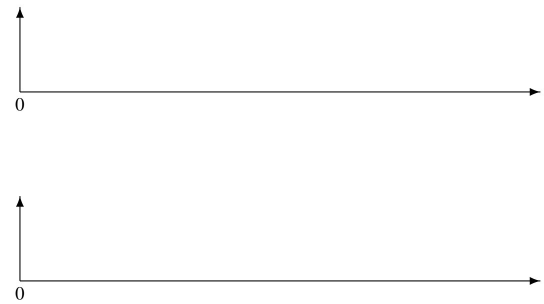

FIG. 14. Latest start times at run-time t = 13.

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., 2016, "A method for handling process overruns and underruns on multiprocessors in real-time embedded systems," *12th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, Auckland, New Zealand, on August 29-31, 2016.

[15] 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.