Effectively Handling Primary and Backup Overruns and Underruns in a Real-Time Embedded System That Tolerates Permanent Hardware and Software Failures

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

***Abstract*—A method and architecture are presented which tolerates both permanent processor failures and permanent software failures by scheduling a primary and a separate backup on different processors for every real-time process, while further increasing system robustness and reliability, by using primary and backup latest start times to effectively handle primary and backup underruns and overruns, *both before, and after, a permanent processor failure*, which significantly increases the chances that a primary or backup may meet its deadline despite permanent hardware failures and permanent software failures, and despite overrunning, while also satisfying additional complex constraints defined on the primaries and backups such as precedence and exclusion relations, in a fault tolerant real-time embedded system.**

I. INTRODUCTION

Real-time embedded systems are used in many safety-critical and complex applications, such as the control of aircraft, automobiles, medical equipment, nuclear reactors and power distribution systems, and all kinds of industrial process control systems. Failures of such systems can have catastrophic consequences such as loss of human life, or massive economic loss. In such systems, a permanent hardware failure may happen when a processor stops functioning, while a permanent software failure may happen when a process produces an incorrect output, or fails to complete before the process deadline.

In order to provide a real-time embedded system with the capability to tolerate a permanent hardware failure, one can provide two versions of programs for each real-time process: a *primary* and a *backup*, and assign every primary and backup that belongs to a same process to different processors, so that whenever a permanent hardware failure on one processor occurs, recovery from the permanent hardware failure can be achieved by abandoning the primaries and backups that were lost due to the permanent processor failure, and executing the corresponding backups and primaries that had been assigned to the different, surviving processors [1, 3, 6, 10]. Similarly, the capability to tolerate permanent software failures can be achieved by aborting the primary, and executing the backup whenever a permanent software failure occurs [6-9]. The backup is often carefully designed to be of less complexity than the primary, producing output of lower, but still acceptable, quality. This is because backups are only occasionally invoked; it impose lighter constraints on the scheduling algorithm if backups take less computation time; also a simpler implementation tends to be more reliable [1].

It is noted that in many systems, including systems which provide a primary and a backup and assign the primary and the backup to different processors, it can be very difficult to accurately estimate the worst-case computation times of real-time primaries and backups due to the widespread use of nondeterministic technologies such as interrupts, DMA, pipelining, caching, prefetching, etc., in order to decrease average-case response times [2, 4], but both overestimating and underestimating worst-case computation times can have very undesirable consequences: overestimating worst-case computation times will cause primaries and backups to underrun and result in low processor utilization; while underestimating worst-case computation times can cause primaries and backups to overrun and cause real-time processes to miss deadlines, which may cause failure of the whole system.

In this paper a method and architecture are presented which further increase system robustness and reliability in fault tolerant real-time embedded systems which provide two versions of each program, a primary and a backup and assign the primary and the backup to different processors, by using primary and backup latest start times to allow primaries or backups to overrun, both before, and after, any permanent processor failure, by effectively utilizing any spare processor capacity including any additional processor capacity created at run-time due to primary or backup underruns, or due to not needing to execute a backup anymore, which significantly increases the chances that a primary or backup may meet its deadline despite permanent hardware failures and permanent software failures, and despite overrunning. The method and architecture presented in this paper also satisfies additional complex constraints defined on the primaries and backups such as precedence and exclusion relations.

Using primaries and backups in a real-time system has been discussed by other authors in [1, 3, 6-10], while handling underruns and overruns has been discussed by other authors in [2, 4, 11] and this author in [14-15]. A significant contribution of the work presented in this paper, is that, to our knowledge, this is the first time that a method and software architecture has been devised that is capable of using primary and backup latest start times to effectively handle primary

and backup overruns and underruns in a real-time embedded system that tolerates both permanent hardware failures and permanent software failures, both before, and after, a permanent processor failure, while also satisfying additional complex constraints defined on the primaries and backups such as precedence and exclusion relations. None of the earlier work, including other authors' work such as [1-11], and this author's earlier work [12-15], include the methods and strategies presented in this paper which significantly increase the chances that a primary or backup will meet its deadline despite permanent hardware failures and permanent software failures, and despite overrunning, *both before, and after, a permanent processor failure*.

II. USING PRIMARIES AND BACKUPS OF REAL-TIME PERIODIC PROCESSES TO PROVIDE THE SYSTEM WTH THE CAPABILITY TO RECOVER FROM HARDWARE AND SOFTWARE PERMANENT FAILURES

It is assumed that each real-time periodic process p is described by a quintuple $(o_p, r_p, c_p, d_p, prd_p)$, where 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. In order to provide a real-time embedded system with the capability to survive and recover from both hardware and software permanent failures, two versions of programs are provided for each real-time process p: (1) a *primary* p_P; and (2) a *backup* p_B.

The following method is used to provide the system with the capability to survive and recover from hardware permanent failures and software permanent failures:

(a) *Before run-time, the primary p_P, and the backup p_B for each process p, are always scheduled to be run on different processors.*

(b) During run-time, when the system hardware and software are functioning normally, only the primary p_P of each process p will be executed. If a primary p_P successfully completes, the corresponding backup p_B will not be executed. The processor time slot reserved for that backup p_B that is not executed can be used to execute other processes' primaries or backups. *Primaries or backups are able to use the time slots for backups that are not executed, or use any unused processor capacity due to primaries or backups underruns, to overrun, thus significantly increasing the chances that primaries or backups will be able to successfully complete before their respective deadlines*.

(c) During run-time, if the primary p_P of each process p suffers a permanent software failure, then the primary p_P of that process will be abandoned; and the backup p_B of that process will be executed, thus guaranteeing that for each process, at least one of either its primary or backup will always be executed in spite of any possible *primary permanent software failure*.

(d) During run-time, if one of the processors suffers a failure, then all the primaries and backups that were scheduled to run on the failed processor will be abandoned; and the primaries and backups on the surviving processors which correspond to a same process of any of the abandoned primaries or backups on the failed processor will be executed, thus guaranteeing that for each process, at least one of either its primary or backup will always be executed in spite of a *permanent hardware failure* of any one of the processors.

III. METHOD FOR COMPUTING A FEASIBLE PRE-RUN-TIME SCHEDULE FOR PRIMARIES AND BACKUPS OF PROCESSES IN WHICH EACH PRIMARY AND EACH BACKUP OF A SAME PROCESS ARE NOT ALLOWED TO BE SCHEDULED ON A SAME PROCESSOR

Procedure for Computing a Feasible Pre-Run-time Schedule for Primaries and Backups in Which Each Primary and backup of a Same Process Are Not Allowed To Be Scheduled on a Same Processor

The procedure below computes a *"feasible-pre-run-time schedule in which each primary p_jP and each backup p_jB of a same process p_j are not allowed to be scheduled on a same processor"* S_O on a multiprocessor, for a set of uncompleted periodic processes P, in order to tolerate hardware failure of a single processor, and in which arbitrary PRECEDES and EXCLUDES relations defined on ordered pairs of processes in P are satisfied.

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

Initially, a PRECEDES relation is defined between each primary p_jP and backup p_jB pair of a same process. The *adjusted deadline* $d'_{p_{j\,PB}}$ for each primary or backup $p_{j\,PB}$ is computed, such that if there does not exists $p_{i\,PB}$ such that $p_{i\,PB}\ PRECEDES\ p_{j\,PB}$ then $d'_{p_{j\,PB}} = d_{p_{j\,PB}}$; else $d'_{p_{j\,PB}}$ = min $\{\ d_{p_{j\,PB}},\ d'_{p_{i\,PB}} - c_{p_{i\,PB}} \mid p_{i\,PB}\ PRECEDES\ p_{j\,PB}\}$.

$t \leftarrow 0$
while $\neg(\forall p_{i\,PB} \in P : \neg(e(p_{i\,PB})) \leq t)$ do
 begin
  for m_q = m_1 to m_N do
   begin
    Among the set
    $\{\ p_{j\,PB} \in P \mid ((\neg(s(p_{j\,PB}) \leq t)$
    $\vee(p_{j\,PB}\ on\ m_q \wedge (s(p_{j\,PB}) < t))$
    % $p_{j\,PB}$ not started yet or $p_{j\,PB}$ started on m_q
    $\wedge(r(p_{j\,PB}) \geq t) \wedge \neg(e(p_{j\,PB}) \leq t)$

% $p_{j\,PB}$ ready and $p_{j\,PB}$ uncompleted
$\wedge\neg(p_{j\,PB} = p_jB \wedge p_jP$ *on* $m_q)$
% if $p_{j\,PB}$ is a backup, then the primary p_jP was
% not previously scheduled on processor m_q
$\wedge(\nexists p_k \in P : (p_k\ EXCLUDES\ p_j)$
$\wedge(s(p_{k\,PB}) < t) \wedge \neg(e(p_{k\,PB}) \leq t))$
% no $p_{k\,PB}$ that has started but not completed
% such that p_k EXCLUDES p_j
$\wedge(\nexists p_k \in P : (p_k\ PRECEDES\ p_j)$
$\wedge\neg(e(p_{k\,PB}) \leq t))$
% no uncompleted $p_{k\,PB}$ such that
% p_k PRECEDES p_j
}
select $p_{j\,PB}$ that has min $d'p_{j\,PB}$.
% earliest-adjusted-deadline-first
in case of ties, select $p_{j\,PB}$ that has a
smaller index number j.
if $\neg(s(p_{j\,PB}) \leq t)$ then $s(p_{j\,PB}) \leftarrow t$.
assign the time unit [t, t + 1] on m_q to $p_{j\,PB}$'s time slot in the pre-run-time schedule S_O.
if the total number of time units assigned to $p_{j\,PB}$'s time slot is equal to $c_{p_{j\,PB}}$, then $e(p_{j\,PB}) \leftarrow t$.
end
$t \leftarrow t\ +\ 1$
end

Example 1.

Fig. 1 shows a feasible pre-run-time schedule S_O for the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I on two processors m_1 and m_2, in which each primary and each backup of a same process are not allowed to be scheduled on a same processor in order to tolerate hardware failure of a single processor, computed by the procedure above. The following EXCLUSION and PRECEDES relations are satisfied: D EXCLUDES I and D PRECEDES I (D PREC I).

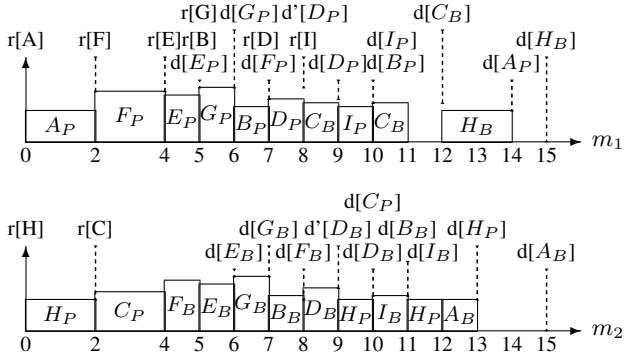

Fig. 1. Feasible pre-run-time schedule S_O for the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I on two processors m_1 and m_2, in which each primary and each backup of a same process are not allowed to be scheduled on a same processor in order to tolerate hardware failure of a single processor, computed by the procedure above.

IV. USING LATEST START TIMES TO EFFECTIVELY HANDLE OVERRUNS AND UNDERRUNS OF PRIMARIES AND BACKUPS IN THE PRESENCE OF PERMANENT HARDWARE AND SOFTWARE FAILURES

A *latest start time LS(p_P) for each primary* p_P, and a *latest start time LS(p_B) for each backup* p_B *in which each primary* p_jP *and each backup* p_jB *of a same process* p_j *are not allowed to be scheduled on a same processor* is determined before and during run-time.

Both before, and after, any permanent processor failure, or permanent software failure, the *latest start time LS(p_P) for each primary* p_P, and a *latest start time LS(p_B) for each backup* p_B *in which each primary* p_jP *and each backup* p_jB *of a same process* p_j *are not allowed to be scheduled on a same processor* will satisfy the following properties:
(1) Every primary p_P and every backup p_B on any surviving processor is guaranteed to be able to start execution on or before its respective latest start time LS(p_P) or LS(p_B);
(2) Every primary p_P and every backup p_B on any surviving processor which starts execution on or before its respective latest start time LS(p_P), and does not fault or overrun, is guaranteed to complete its computation on or before its process deadline d(p);
(3) If any primary p_P or any backup p_B on any surviving processor overruns, that is, does not complete after executing for a number of time units equal to its worst-case computation time, then that primary or backup can continue to execute, as long as no other primary or backup with an earlier deadline is prevented from starting on or before its latest start time, while guaranteeing that every other primary $p_{i\,P}$ and every other backup $p_{i\,B}$ will still be able to:
(3a) start execution on or before their respective latest start times LS($p_{i\,P}$) or LS($p_{i\,B}$);
(3b) complete their computations on or before their respective deadlines d($p_{i\,P}$) or d($p_{i\,B}$) as long as they do 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 backup underruns, or due to not needing to execute a backup any more, in order to increase the chances that either the primary or the backup of each process will be able to successfully complete its computation before its deadline, even after a processor failure has occurred.

Procedure for Computing a Latest Start Time Schedule and Latest Start Times for Primaries and Backups in Which Each Primary and Backup of a Same Process Are Not Allowed To Be Scheduled on a Same Processor

The following procedure, when given an original feasible pre-run-time schedule in which each primary p_jP and each backup p_jB of a same process p_j are not allowed to be scheduled on a same processor S_O, computes a latest-start-time schedule S_L in which each primary p_jP and each backup p_jB of a same process p_j are not allowed to

be scheduled on a same processor, by scheduling all the primaries and all the backups p_{PB} in each process p in P starting from time t equal to the latest deadline among all the primaries and all the backups p_{PB} in P, that is, t = max $\{d_p \mid \forall p_{PB} \in \text{P}\}$, in reverse time order, using a "Latest-Release-Time-First" scheduling strategy that is equivalent to a reverse application of the well known Earliest-Deadline-First strategy, which satisfies all the PREC relations defined below:

Given any original feasible pre-run-time schedule S_O on a multiprocessor, we first define a set of "PREC" relations on primaries and backups p_{PB} in the set of processes p in P in the feasible pre-run-time schedule S_O:
$\forall p_{iPB} \in P, p_{jPB} \in P$,
if $e(p_{iPB}) < ep_{jPBPB}) \wedge$ ((p_i EXCLUDES p_j) $\vee$ (p_{iPB} PRECEDES p_{jPB}))
then let p_{iPB} PREC p_{jPB}

Let the set of processors be $M = \{m_1, \ldots, m_q, \ldots, m_N\}$. In the procedure below, "p_j on m_q in S_O" means "process p_j is scheduled on processor m_q in the original feasible pre-run-time schedule S_O". "$LS(p_{jPB})$" refers to the "latest-start-time" of p_{jPB}, or the beginning (left hand side) of p_{jPB}'s time slot in the the newly constructed latest-start-time schedule S_L, which is also equal to the time value of the left boundary of the last time unit [t − 1, t] that will be assigned by the procedure to p_{jPB}'s time slot while constructing the latest-start-time schedule S_L. "$e(p_{jPB})$" refers to the "end time" of p_{jPB}, or the end (right hand side) of p_{jPB}'s time slot in the newly constructed latest-start-time schedule S_L.

Initially, compute the *adjusted release time* $r'_{p_{jPB}}$ for each primary or backup p_{jPB} where if there does not exists p_{iPB} such that p_{iPB} $PREC$ p_{jPB} then $r'_{p_{jPB}}$ = $r_{p_{jPB}}$; else $r'_{p_{jPB}}$ = max $\{$ $r_{p_{jPB}}$, $r'_{p_{iPB}}$ + $c_{p_{iPB}}$ | p_{iPB} $PREC$ $p_{jPB}\}$.

$t \leftarrow$ max $\{d_p \mid \forall p \in \text{P}\}$
while $\neg(\forall p_{iPB} \in P : \neg(LS(p_{jPB}) \geq t)$ do
begin
for m_q = m_1 to m_N do
begin
if ($\exists p_{iPB} : p_{iPB}$ *on* m_q *in* S_O
$\wedge(t = d_{p_{iPB}} \vee t = LS(p_{iPB})))$
% if for some p_{iPB} on m_q in S_O
% t is equal to the deadline of p_{iPB} or t is equal to
% the latest-start-time of p_{iPB} on m_q in S_L
then
begin
Among the set
$\{$ $p_{jPB} \mid p_{jPB}$ *on* m_q *in* S_O
$\wedge(t \leq d_{p_{jPB}} \wedge \neg(LS(p_{jPB}) \geq t)$
% Among the set of p_{jPB} on m_q in S_O such that
% t is less than or equal to the deadline of p_{jPB}
% or t is less than or equal to the latest-start-time
% of p_{jPB} on m_q in S_L and
$\wedge(\nexists p_{kPB} \in P :$
p_{jPB} PREC $p_{kPB} \wedge \neg(LS(p_{kPB}) \geq t))$
% no p_{kPB} such that t is greater than the start
% time of p_{kPB} on m_q in S_L and p_k PREC p_j
$\}$
select p_{jPB} that has max $r'_{p_{jPB}}$.
% latest-adjusted-release-time-first
in case of ties, select p_{jPB} that has a
greater index number j.
if $\neg(e(p_{jPB}) \geq t)$ then $e(p_{jPB}) \leftarrow t$.
assign the time unit [t − 1, t] on m_q to p_{jPB}'s
time slot in the latest-start-time schedule S_L.
if the total time units assigned to p_{jPB}'s time
slot is equal to $c_{p_{jPB}}$, then $LS(p_{jPB}) \leftarrow t$ - 1.
end
else if the time unit [t, t + 1] on m_q was previously
assigned to some p_{kPB} and t $\neq LS(p_{kPB})$ then
begin
assign the time unit [t − 1, t] on m_q to p_{kPB} in
the latest-start-time schedule S_L.
if the total time units assigned to p_{kPB}'s time
slot is equal to $c_{p_{kPB}}$, then $LS(p_{kPB}) \leftarrow t$ - 1.
end
end
$t \leftarrow t$ - 1
end

Example 2.

Fig. 2 below shows a latest-start-time schedule S_L and the latest start times for all the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I on two processors m_1 and m_2, in which each primary and each backup of a same process are not allowed to be scheduled on a same processor, which can be computed by the procedure above from the feasible pre-run-time schedule S_O in Fig. 1, in which D PREC I is satisfied.

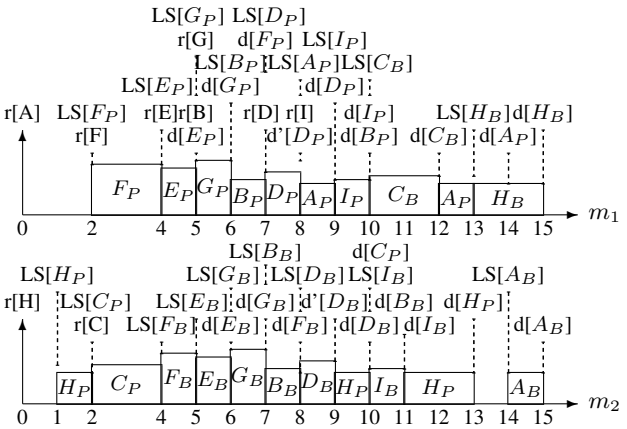

Fig. 2. Latest-start-time schedule S_L and the latest start times for all the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I on two processors m_1 and m_2, in which each primary and each backup of a same process are not allowed to be scheduled on a same processor in order to tolerate hardware failure of a single processor, and which satisfies D PREC I, computed by the procedure above.

V. HANDLING PRIMARY AND BACKUP OVERRUNS AND UNDERRUNS WHILE TOLERATING PERMANENT HARDWARE AND PERMANENT SOFTWARE FAILURES AT RUN-TIME

In the following, before a permanent processor failure, all the processors are called "surviving processors"; after a permanent processor failure, all the processors except the failed processor are called "surviving processors."

5.1. Possible Primary and Backup Recovery Schemes After A Permanent Processor Failure

After a permanent processor failure, there exist at least the following possible recovery schemes concerning which entity should be executed on a surviving processor to replace each primary $p_{i\,P}$ or each backup $p_{j\,B}$ that was previously scheduled to execute on the failed processor:

(1) Primary Recovery Scheme A: For each primary $p_{i\,P}$ that was previously scheduled to execute on the failed processor, use the existing corresponding backup $p_{i\,B}$ on a surviving processor for recovery.

(2) Primary Recovery Scheme B: For each primary $p_{i\,P}$ that was previously scheduled to execute on the failed processor, try to execute that same primary $p_{i\,P}$ on a surviving processor for recovery.

(3) Backup Recovery Scheme A: For each backup $p_{j\,B}$ that was previously scheduled to execute on the failed processor, use the corresponding existing primary $p_{j\,P}$ on a surviving processor for recovery.

(4) Backup Recovery Scheme B: For each backup $p_{j\,B}$ that was previously scheduled to execute on the failed processor, if a feasible schedule in which all timing constraints are satisfied exists for that same backup $p_{j\,B}$ together with all other surviving primaries and backups that are used for recovery on the surviving processor(s), then use that same backup $p_{j\,B}$ to replace the corresponding primary $p_{j\,P}$ on a surviving processor for recovery, otherwise use the corresponding existing primary $p_{j\,P}$ on a surviving processor for recovery. .

The main advantages and disadvantages of the above schemes can be summarized as follows:

With scheme (1) and (3), the existing feasible pre-run-time schedule, latest start time schedule, and latest start times can be used without significant changes to guarantee that all the surviving primaries and backups will be able to meet their deadlines.

With scheme (1) and (3), each processor would also only need to be able to access the copies of the primaries and backups that were assigned to that processor in the existing latest start time schedule.

In contrast, with scheme (2) and (4), each processor would also need to always be able to access copies of all the primaries and copies of all the backups, including the primaries and backups that were assigned to other processors in the existing latest start time schedule, in case any one of the processors fail.

If scheme (4) can be used, the system reliability could be increased compared with using scheme (3), as simpler backups tend to be more reliable than primaries, but using scheme (4) may require recomputing the latest-start-time schedule, and using scheme (4) instead of scheme (3) may not always be possible.

With scheme (2) and (3), the primary $p_{i\,P}$ used to replace the backup $p_{i\,B}$ will likely be of higher quality but of less reliability compared with the replaced backup $p_{i\,B}$.

With scheme (1) and (4), the backup $p_{j\,B}$ used to replace the primary $p_{j\,P}$ will likely be of less quality but of higher reliability compared with the replaced primary $p_{j\,P}$.

A major issue with scheme (2) is that the existing feasible pre-run-time schedule, latest start time schedule, and latest start times cannot be used anymore to guarantee that all the surviving primaries and backups will be able to meet their deadlines, because the length of the time slot reserved for the backup in the existing latest start time schedule can be shorter than the length needed to execute the primary. With scheme (2), one would need to pre-compute alternative feasible pre-run-time schedules, latest start time schedules, and latest start times that cover all the possible processor failure scenarios, in order to determine whether all primaries and backups will still be able to meet their deadlines when any processor fails.

In the following, we will assume that scheme (1) and scheme (3) are used, because of their advantages:

(a) Scheme (1) and scheme 3 are *the simplest to implement*, since the existing feasible pre-run-time schedule and latest start time schedule and latest start times can be used without significant changes to guarantee that all the surviving primaries and backups will be able to meet their deadlines after any permanent processor failure;

(b) Scheme (1) and scheme 3 *require the least amount of redundant information, memory, and communication resources*, since each processor would also only need to be able to access the primaries and backups that were assigned to that processor in the existing latest start time schedule.

However, it is emphasized here that using scheme (4) instead of scheme (3) is also a very valid approach since scheme (4) may increase system reliability.

5.2. Tolerating Permanent Processor Failures and Permanent Software Failures

With one backup for each primary, the method presented here provides the capability to recover from one permanent processor failure, as well as recover from any number of permanent primary software failures.

Note that the assumption that there is only one backup for each primary, implies that the system can only recover from one permanent processor failure that does not occur in conjunction with any software failure. If (i) the capability to recover from more than one permanent processor failure, and/or (ii) the capability to recover from a permanent processor failure that occurs in conjunction with a software failure are required, then a higher level of processor and process backup redundancy [5], that is, more than two

processors and more than one backup for each real-time process will be required.

Run-Time Scheduler Method For Effectivly Handling Primary and Backup Overruns and Underruns While Tolerating Permanent Processor Failures and Permanent Software Failures

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 a permanent processor failure has occurred.

(b) At a time t when some primary p_P or backup p_B 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 $LS(P_B)$ of some backup p_B.

(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 backup p_B will be activated, and the primary p_P will be aborted.

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

Let t be the current time.

Step 0. In situation (a) above, when a permanent processor failure has occurred, each primary p_{iP} that was previously scheduled to execute on the failed processor will be abandoned, and the existing corresponding backup p_{iB} on a surviving processor will be activated for recovery: let $ActivationTime(p_{iB})$ = t. Each backup p_{jB} that was previously scheduled to execute on the failed processor will be abandoned, and the corresponding surviving primary $p_{iP'}$ on a surviving processor will be activated for recovery: let $ActivationTime(p_{iP'})$ = t.

After a permanent processor failure, we call any primary p_{iP} for which the corresponding backup p_{iB} was previously scheduled before run-time on the permanently failed processor, a *"surviving primary"*, and denote it as "$p_{iP'}$". Any surviving primary $p_{iP'}$ will have the same deadline that its corresponding backup p_{iB} had on the permanently failed processor, when re-computing latest start times, and when the run-time scheduler selects primaries and backups for execution on each surviving processor at run-time. Any surviving primary $p_{iP'}$ will be given the same priority as a backup when the run-time scheduler selects primaries and backups for execution on each surviving processor at run-time.

Recompute the latest start times for the primaries and backups on each surviving processor .

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 backup p_B has generated a fault. If so perform error handling.

Step 1. In situation (f) above, if a primary p_P generates a permanent software failure, then the primary p_P will be aborted, and the corresponding backup p_B will be activated; let $ActivationTime(p_B)$ = t.

Step 2. 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 a backup p_B has been reached, that is, LS(p_B) = t, then the primary p_P will be aborted, and the corresponding backup p_B will be activated; let $ActivationTime(p_B)$ = t.

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

Step 3. 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 backup p_B at time t; let $ActivationTime(p_B)$ = t.

Recompute the latest start time $LS(p_P)$ or $LS(p_B)$ for each uncompleted primary p_P or backup p_B that was executing at time $t-1$ and has not overrun at time t. Note that once the initial latest start times have been computed before run-time, at run-time the run-time overhead can be significantly reduced by only recomputing the latest start time $LS(p_P)$ or $LS(p_B)$ for each uncompleted primary p_P or backup p_B that had just been preempted that was executing at time $t-1$ using the method described in [14].

Run-Time Scheduler Method: *Before and after a permanent processor failure, the run-time scheduler method selects primaries and backups for execution on each surviving processor m_q at run-time in the following priority order:*

Priority 0 Tasks: The highest priority is given to any backup p_B or any surviving primary $p_{iP'}$, such that the latest start time of p_B or $p_{iP'}$ has been reached, that is, $LS(p_B) = t$ or $LS(p_{iP'}) = t$; or any backup p_B or any surviving primary $p_{iP'}$ such that p_B has been activated or $p_{iP'}$ has been activated; and has the earliest adjusted deadline $d'(p_B)$ or $d'(p_{iP'})$ among all such tasks on m_q that are ready and have not completed at time t. This is because successful completion of backup p_B before its adjusted deadline $d'(p_B)$ or surviving primary $p_{iP'}$ before its adjusted deadline $d'(p_{iP'})$, is considered to be the *"last chance to avoid failure of the task/process "* p or p_i, and potentially, the *"last chance to avoid failure of the entire system."*

Priority 1 Tasks: The next highest priority is given to any primary p_P such that the latest start time of primary p_P has been reached, that is, $LS(p_P) = t$, and p_P has the earliest adjusted deadline d'_{pP} among all such tasks on m_q that are ready at time t.

Priority 2 Tasks: The next highest priority is given to any primary p_P such that p_P is ready at time t and p_P has the earliest adjusted deadline d'_{pP} among all such tasks on m_q that are ready at time t.

Step 4. 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 5. Let the primaries p_P or backups p_B that were selected in Step 3 start to execute at run-time t.

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 m_1 and m_2 of the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I shown in Fig. 1 of Example 1, assuming that processor m_1 will suffer a permanent hardware failure at time t = 4.5, but for every process in the set of processes A, B, C, D, E, F, G, H, I either a primary or a backup will still be able to complete on processor m_2 before their deadlines despite overrunning or underrunning. The latest start time values s of the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I shown in Fig. 2 in Example 2 will be used at run time t = 0. In Fig. 3, F_P underruns, while A_P, C_P, H_P overruns. The portions of the run-time execution during which A_P, C_P, H_P overruns are shown using dashed lines.

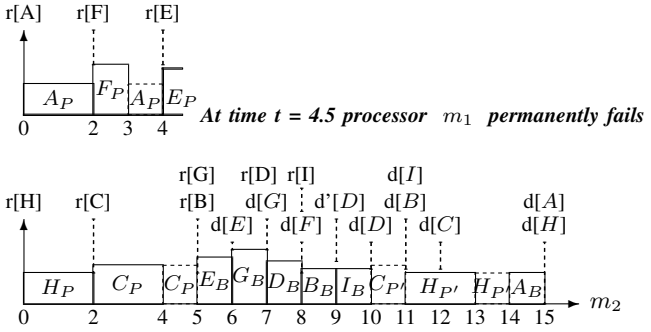

Fig. 3. Run-time execution on two processors m_1 and m_2 of the primaries and backups in the set of processes A, B, C, D, E, F, G, H, I. Processor m_1 permanently fails at time t = 4.5.

At run-time t = 0: the latest start time schedule is shown in Fig. 3. The run-time scheduler will select primary A_P and primary H_P to run on processor m_1 and processor m_2 respectively as priority 2 tasks, because A and H are the processes that are ready with the earliest adjusted deadline at time t = 0. At t = 0, the timer will be programmed to interrupt at F_P and C_P's latest start times LS(F_P) = LS(C_P) = 2, before dispatching A_P and H_P for execution.

At time t = 2: the timer interrupts at F_P and C_P's latest start times LS(F_P) = LS(C_P) = 2; while both H_P and A_P have not completed and A_P overruns. After re-computing the latest-start-times, LS(H_P) = 11, and A_P's time slot is removed from the latest-start-time schedule. The run-time scheduler will first select primary F_P to run on processor m_1 as a priority 1 task, because primary F_P's latest start time LS(F_P) = 2 has been reached. Then the run-time scheduler will select primary C_P to run on processor m_2 as a priority 1 task, because primary C_P's latest start time LS(C_P) = 2 has also been reached. At t = 2, the timer will be programmed to interrupt at primary E_P's latest-start-time LS(E_P) = 4, before dispatching F_P and C_P for execution.

At time t = 3: primary F_P underruns, while C_P has not completed. After re-computing the latest-start-time for C_P at time 3, LS(C_P) = 7, and backup F_B's time slot will be removed from the latest-start-time schedule. The run-time scheduler will select A_P to run on m_1 as a priority 2 task. *Note that A_P is able to use the portion of the time slot that is unused due to primary F_P's underrun to overrun.* The run-time scheduler will select C_P to run on m_2 as a priority 2 task, because C_P has the earliest deadline among all tasks on m_2 that are ready at time t = 3. At t = 3, the timer will be programmed to interrupt at primary E_P's latest start time LS(E_P) = 4, before dispatching A_P and C_P for execution.

At time t = 4: primaries A_P and C_P have not completed, and C_P overruns.

After re-computing the latest-start-times, C_P's time slot is removed from the latest-time schedule. The run-time scheduler will select primary E_P to run on processor m_1 as a priority 1 task, because primary E_P's latest start time LS(E_P) = 4 has been reached.

The run-time scheduler will select primary C_P to start overrunning on m_2 as a priority 2 task. *Note that C_P is able to use the time slot that was previously reserved for backup F_B to overrun - F_B's time slot is not used anymore due to the underrun of primary F_P at time 3.* At t = 4, the timer will be programmed to interrupt at primary G_P's and backup E_B's latest start time LS(G_P) = LS(E_B) = 5, before dispatching E_P and C_P for execution.

At time t = 4.5 processor m_1 permanently fails.

Primaries A_P, E_P that previously were in execution and had not completed yet on the failed processor m_1 at time $t = 4.5$ will be aborted and the corresponding backups A_B, E_B on the surviving processor m_2 will be activated.

Note that all the surviving primaries, that is, $H_{P'}$ and $C_{P'}$, on the surviving processor m_2 for which the corresponding backups H_B, C_B were previously scheduled before run-time on the permanently failed processor m_1, will be given the same priority as a backup when the run-time scheduler selects primaries and backups for execution on the surviving processor m_2.

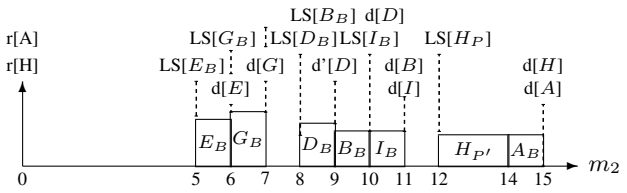

FIG. 4. Latest start times at run-time t = 4.5 after m_1 has failed.

After processor m_1 has failed, all the primaries and backups on processor m_1 will cease to exist. Because H_B does not exist anymore, H_P is not constrained to complete

before H_B anymore, so the latest start time for H_P, LS(H_P) = 12, as shown in Fig. 4.

At time t = 5: survivor primaries $H_{P'}$ and $C_{P'}$ have not completed yet, while backup E_B's earliest start time LS(E_B) = 5 has been reached. The recomputed latest start times are shown in Fig. 4. The run-time scheduler will select backup E_B to run on processor m_2 as a priority 0 task, because backup E_B's latest start time LS(E_B) = 5 has been reached, and it has the earliest adjusted deadline among all backups or surviving primaries for which their latest start times have been reached or have been activated on m_2 at time t = 5. At t = 5, the timer will be programmed to interrupt at backup G_B's latest start time LS(G_B) = 6, before dispatching E_B for execution.

At time t = 6: backup E_B completes, while backup G_B's earliest start time LS(G_B) = 6 has been reached. The run-time scheduler will select backup G_B to run on processor m_2 as a priority 0 task, because backup G_B's latest start time LS(G_B) = 6 has been reached, and it has the earliest adjusted deadline among all backups or surviving primaries for which their latest start times have been reached or have been activated on m_2 at time t = 6. At t = 6, the timer will be programmed to interrupt at backup D_B's latest start time LS(D_B) = 8, before dispatching G_B for execution.

At time t = 7: backup G_B completes. The run-time scheduler will select backup D_B to run on processor m_2 as a priority 0 task. At t = 7, the timer will be programmed to interrupt at backup B_B's latest start time LS(B_B) = 9.

At time t = 8: backup D_B completes. The run-time scheduler will select backup B_B to run on processor m_2 as a priority 0 task. At t = 8, the timer will be programmed to interrupt at backup I_B's latest start time LS(I_B) = 10.

At time t = 9: backup B_B completes. The run-time scheduler will select backup I_B to run on processor m_2 as a priority 0 task. At t = 9, the timer will be programmed to interrupt at survivor primary $H_{P'}$'s latest start time LS($H_{P'}$) = 10.

At time t = 10: backup I_B completes. The run-time scheduler will select survivor primary $C_{P'}$ to run on processor m_2 as a priority 0 task. At t = 10, the timer will be programmed to interrupt at survivor primary $H_{P'}$'s latest start time LS($H_{P'}$) = 12.

At time t = 11: *survivor primary $C_{P'}$ is able to complete before its deadline, after the permanent hardware failure of processor m_1, despite overrunning.* The run-time scheduler will select survivor primary $H_{P'}$ to run on processor m_2 as a priority 0 task. At t = 11, the timer will be programmed to interrupt at backup A_B's latest start time LS(A_B) = 14.

At time t = 14: *survivor primary $H_{P'}$ is also able complete before its deadline, after the permanent hardware failure of processor m_1, despite overrunning.* The run-time scheduler will select backup A_B to run on processor m_2 in Step A as a priority 0 task. At t = 14, the timer will be programmed to interrupt at backup A_B's deadline d_{A_B} = 15.

At time t = 15: backup A_B completes before its deadline.

VI. CONCLUSIONS

In this paper a method and architecture were presented which further increase system robustness and reliability in fault tolerant real-time embedded systems which provide two versions of each program, a primary and a backup and assign the primary and the backup to different processors, by using primary and backup latest start times to allow primaries or backups to overrun, **both before, and after, a permanent processor failure**, by effectively utilizing any spare processor capacity including any additional processor capacity created at run-time due to primary or backup underruns, or due to not needing to execute a backup anymore, to significantly increase the chances that a primary or backup will meet its deadline despite permanent hardware failures and permanent software failures, and despite overrunning. The method and architecture presented in this paper also satisfies additional complex constraints defined on the primaries and backups such as precedence and exclusion relations.

REFERENCES

[1] Krishna, C.M., 2014, "Fault-tolerant scheduling in homogeneous real-time systems." *ACM Computing Surveys*., vol. 46, no. 4, pp. 134, Apr. 2014.

[2] 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.

[3] Bertossi, A.A., Mancini, L.V., and Rossini, F., 1999, "Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems." *IEEE Trans. Parallel and Distr. Sys.*, Vol. 10, No. 9, pp. 934-945, September 1999.

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

[5] 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.

[6] Bertossi, A.A., Mancini, L.V., and Menapace, A., 2006, "Scheduling hard-real-time tasks with backup phasing delay." *IEEE Symp. on Distributed Simulation and Real-Time Applications (DS-RT)*, 2006.

[7] Koren, I., and Krishna, C.M., 2007, "Fault-Tolerant Systems." Morgan-Kaufman, 2007.

[8] Krishna, C.M., and Shin, K.G., 1996, "Scheduling tasks with a quick recovery from failure." *IEEE Trans. on Computer*, vol. C-35, no. 5, May. 1986.

[9] Pradhan, D.K., 1996, "Fault-Tolerant Computer System design." Prentice Hall, 1996.

[10] Siewiorek, D., and Swarz, R., 1999, "Reliable Computer Systems: Design and Evaluation." A.K. Peters, 1999.

[11] 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.

[12] 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.

[13] 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.

[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.