Efficient Mode Changes in Multi-Mode Systems

Akramul Azim
Department of Electrical, Computer and Software Engineering
University of Ontario Institute of Technology
Ontario, Canada
Email: akramul.azim@uoit.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
Email: sfischme@uwaterloo.ca

***Abstract*—Multi-mode systems work in configurations, but face the challenge of ensuring timing guarantees during mode changes. In a multi-mode system, a mode-change request occurs when the system wants to operate in a new mode, but is already running in one. One mode may include some tasks that are same as that of another mode. Therefore, the new mode may have tasks that are same as the old mode. Changing modes in such a way to skip some already completed tasks can decrease the workload of the new mode.**

Traditional protocols for changing modes always look forward in time to schedule tasks, although using already completed tasks may avoid re-executing them in the new mode. Reusing common tasks reduces the time to re-execute them while switching modes. In this paper, we introduce the concept and design considerations for a mode-change technique that may use completed tasks stored in checkpoints to avoid unnecessary re-execution and facilitate faster execution of new mode tasks. Through an example case-study, experimental results demonstrate that the overhead of using checkpoints is low, and using rollback facilitates faster execution of new mode tasks if completed tasks stored in checkpoints can be reused.

I. Introduction

Many systems that control physical objects run in multiple modes. Systems require to operate in multiple configurations for different functionalities and also to provide flexibility. For example, an automobile system can have a particular configuration or mode for stopping, starting, or cruising. The transitions between modes can happen dynamically at run time and are sensitive to the control of a system because of the mode-change delay incurred during mode changes. Moreover, some systems can have multiple modes of operation and quality-of-service depends on the delay incurred during mode changes.

A multi-mode system switches between modes when a mode-change request (MCR) takes place. Systems operating in different modes such as the initialization mode, an emergency mode, and a fault-recovery mode may exhibit multiple behaviours. In this work, we consider that a multi-mode system has a set of tasks, which are comprised of the following: (1) old mode tasks, (2) new mode tasks, and (3) common tasks. An old mode task belongs to the old mode while switching to a new mode and is different than a new mode task. A new mode task can either be a modified old task or a completely new task. A common task has the same temporal behavior (e.g., period and deadline) both in the old mode and the new mode. A common task exists in different modes but can avoid executing while switching modes. For example, a fault-tolerant video monitoring system has two modes: high and low quality. Switching from the low to high quality mode may require executing (1) old mode tasks such as the low-quality transmissions, (2) common tasks can include taking inputs periodically from cameras, calculating average power usages, or gathering information on faulty memory blocks, (3) new mode tasks such as transmitting high-quality transmissions will execute once switching to the new mode is completed.

During a mode change, the running tasks of the old mode and the tasks of the new mode can use any of the two types of protocols: synchronous and asynchronous. In a synchronous mode-change protocol, tasks of the old mode that are ready to run finish executing before the new mode tasks are released. However, an asynchronous mode-change protocol allows scheduling the tasks of the old and new modes based on their priorities.

To the best of our knowledge, all work on multi-mode systems only considered mode-change protocols that require the system to move forward in time (i.e., the system makes progress). These work lack the concept that a system might be able to undo recent activities to reach faster to the new mode. This paper proposes a novel idea to use execution information of prior successful completed tasks stored in checkpoints when they are suitable for faster mode changes. For some mode changes, using prior completed tasks is inefficient because of either unsupported rollback or high overhead. The proposed mode-change model uses existing known protocols (i.e., minimum single-offset protocols) for the proof of concept, but can be applied to other mode-change protocols.

A checkpoint in computing refers to a state of the execution of the system when it is saved. The state of execution refers to a snapshot of the system at a certain time. Using checkpoints, it is possible to reinstate a system from the saved state. For a large system, saving the state of the system at each checkpoint is memory intensive. To address this issue, a system can use checkpoints to perform like atomic operations with negligible overhead using copy-back cache and atomic update feature of stable transaction memory (STM). Checkpoints are usually used to provide fault-tolerance [13]. A system may re-execute from a saved state for recovery upon detecting a fault or an error. The method of using checkpoints for fault-tolerant systems is popular and exists in many implementations [9], [13], [4], [8]. However, in this paper, we propose to use the

concept of checkpoints in a multi-mode system.

Using checkpoints in changing modes can either avoid re-executing successfully completed common tasks when switching to a different mode or the same mode insisting on undoing recent activities. A system uses prior executions of the already completed tasks stored at a checkpoint to utilize system resources efficiently. Therefore, this work aims to model a system that permits the transition from the old mode to the new mode through the use of a checkpoint when an MCR occurs. For example, if a system has two modes and they have common tasks having earlier deadlines than other tasks, then while changing modes the system will be able to use a checkpoint if required that has already stored the completed executions of the common tasks.

Using checkpoints for mode changes imposes a number of requirements: (1) it is necessary to perform a schedulability test to ensure that the checkpoints must not cause the tasks of any mode to miss their deadlines, (2) it is required to derive a bound on the number of checkpoints to keep the number of checkpoints finite, (3) it is necessary to derive a relationship between the bound on the number of checkpoints and the number of mode changes, because a system cannot allow an infinite number of mode changes in each hyperperiod, and (4) to ensure efficiency and flexibility, the system should avoid using a checkpoint if a better time-efficient transition is available to switch from the old mode to the new mode upon an MCR.

The design flow for the system to use rollback-supported mode changes is as follows: (1) the system designer derives the necessary overhead associated with a checkpoint for storing and retrieving variables, (2) according to a synchronous or an asynchronous mode-change protocol, a method calculates the worst-case mode-change delay of the tasks, (3) using the schedulability analysis conditions for a particular scheduling policy and a number of allowable mode changes, a method derives the specifications of a checkpoint to use as a periodic task and the bound on the number of checkpoints, (4) the implementation of system contains synchronization protocols to ensure consistency while running the schedule in an application.

To allow faster mode changes by reducing the time required for re-executing common task, this paper presents a rollback-based mode-change mechanism. This paper also presents:

- analysis of the mode-change delay in single processor or multiprocessors with or without checkpoints for synchronous and asynchronous minimum single-offset protocols,
- analysis of the periodic specifications of a checkpoint to derive a bound on the number of checkpoints, and
- analysis for using checkpoints in multi-mode synchronous and asynchronous minimum single-offset protocols.

The remainder of this paper starts with Section II, which discusses some related work. Section III presents the system model and assumptions. Section IV-A discusses the workflow for using checkpoints in multi-mode systems such that the timing and synchronization requirements are met. Section IV-B discusses checkpoint-based minimum single offset protocols. Section IV-C provides analysis of mode-change delay for using checkpoints in synchronous and asynchronous multi-mode systems. Section IV-D explains how to derive a bound on the number of checkpoints. Section V demonstrates the applicability of using checkpoints in multi-mode systems through experimental analysis. Section VI concludes the paper.

II. Related Work

Multi-mode real-time systems in a uniprocessor platform have been studied extensively in [16], where only one shared processor is available to execute all tasks. Some recent works such as [11], [22] on multi-mode real-time systems have focused on multiprocessor systems, where tasks can be executed on several shared processors. A semantic framework for mode-change protocols has also been proposed in [12].

In [16], Real and Crespo survey some mode-change protocols and categorize them between synchronous and asynchronous protocols. A synchronous protocol avoids executing the new mode tasks until all the old mode running task finish their execution. An asynchronous protocol may have a less mode-change delay than a synchronous protocol because of scheduling new mode and old mode tasks together.

Tindell and Alonso [21] propose the idle time protocol, in which the system allows execution of background tasks besides periodic and sporadic tasks. This protocol uses the operating system to perform the mode changes. However, some protocols [17] allow the mode-change operation to be performed on controllers that have no operating system.

Nelis et al. [11], [22] propose mode-change protocols for scheduling multi-mode real-time systems on a multiprocessor platform. They propose two variations of the minimum single offset protocol [16] for multiprocessor systems: synchronous and asynchronous. The synchronous minimum single offset protocol for multiprocessors avoids scheduling tasks of the old mode and the new mode simultaneously that the asynchronous minimum single offset protocol can do.

Real-time systems use checkpoints for fault tolerance, but not for mode changes. Several works exist on using checkpoints to recover from faults. Koo and Toueg describe two ways [9] of setting checkpoints: (1) each station locally and independently creates checkpoints, and (2) all stations create global checkpoints. The scheme allows stations to place checkpoints dynamically. To protect systems from failures, Bertossi et al. propose two types of recoveries [4]: (1) a static schedule for recovery that consists of checkpoints at periodic intervals, and (2) a dynamic schedule for recovery that places checkpoints at each slot. Zhang and Chakrabarty present techniques [24] to find out a checkpoint interval that is used to insert checkpoints equidistantly for tolerating a bounded number of faults. Chen and Ren determine the number of checkpoints [8] for soft real-time systems based on the task execution time and the task deadline miss probability. They establish a mathematical model to find the optimal checkpoint interval for soft real-time systems with the assumption that all application tasks have the same priority. Izosimov et al. use both re-execution and replication mechanisms [13] as fault-tolerance techniques to deal with the transient faults.

Checkpoints are used for re-execution and placed at equidistant intervals in a static schedule to tolerate faults.

A real-time system has to perform schedulability analysis prior using checkpoints. In [15], the authors propose an optimization framework to obtain an optimal number of checkpoints for a fixed priority preemptive scheduling scheme. Bowen and Pradhan [5] provide an analysis on a processor and memory-based checkpointing schemes which allow checkpoints to perform like atomic operations with negligible overhead using copy-back cache and atomic update feature of STM.

Our work aims to use rollback to a checkpoint to avoid re-executing any common tasks. This mechanism can be applied to any previous related work on multi-mode real-time systems. Therefore, the related work on checkpoints for fault-tolerance can still be applied together with the idea of avoiding unnecessary executions if rollback is being used for mode changes.

III. Model of Computation

Assume that a multi-mode real-time system has n number of operating modes $M_1, M_2, \ldots, M_n$. Each operating mode M_k has to execute a task set $T = \{\tau_1, \tau_2, \ldots, \tau_s\}$ containing s number of independent tasks. Each task $\tau_i = (p_i, d_i, e_i)$ is characterized by three parameters: the period p_i, the deadline d_i, and the worst-case execution time e_i. Assume that e_i, p_i, and d_i for each task τ_i are known *a priori*.

This work assumes the use of either single processor or identical multiprocessors. An identical multiprocessor platform has some processors having the same properties such as a uniform memory architecture and execution speed. Therefore, the processors are interchangeable. This work also assumes the existence of a global clock or a clock synchronization protocol as discussed, for instance, in [10], [23], [14]. A clock synchronization protocol adjusts the offsets caused by clock drifts.

Mode-change requests are common in multi-mode systems. An event causes an $\mathrm{MCR}(x, y)$ to switch from the current mode (i.e., old mode) M_x to a new mode M_y where $x, y \in \mathbb{N}^+$. At the time of an MCR, the tasks that are ready to run in the old mode complete their execution, and the system disables the release of new instances of tasks from the old mode. This work also assumes that at most one MCR is active at any time. The modes of the system are scheduled using rate monotonic (RM) scheduling algorithm for a single processor or partitioned RM for multiprocessors, leaving the extension to other scheduling schemes such as earliest deadline first (EDF) for future work.

This work assumes that common tasks exist between any two modes of the system, leaving the extension of varying common tasks between different mode changes for the future. This work also assumes that an MCR cannot be requested during the transition between two modes. Depending upon the types of application, a system decides whether to continue execution of the current task or to abort it, because task dropping is usually performed when it is feasible without loss of data consistency. This work assumes that every task of the old mode must complete its execution when an MCR occurs.

Definition 1 (Checkpoint-based multi-mode system properties)**.** *A checkpoint-based multi-mode system has the following properties:*

- *W determines the workload consisting of a set of modes and a scheduling policy.*
- *$M = M_1, \ldots, M_n$ is the set of n modes.*
- *T is the set of all tasks in the system.*
- *$T_m = (\tau_1^m, \ldots, \tau_s^m)$ is the set of tasks in mode $m \leq n$ and $s \in \mathbb{N}^+$.*
- *Each task in T_m has an execution time (e_i^m), a deadline (d_i^m), and a period (p_i^m) and is characterized by (p_1^m, d_1^m, e_1^m).*
- *$T_o \subset T$ denotes the set of old mode tasks.*
- *$T_w \subset T$ denotes the set of new mode tasks.*
- *T_u denotes the set of common tasks such that $T_u \subset T_o$ and $T_u \subset T_w$.*
- *A checkpoint is a task $\tau_c \in T$ with period p_c. The number of checkpoints is the number of instances of periodic task τ_c. For example, checkpoint τ_c^w will denote instance w of task τ_c such that $w \in \mathbb{N}^+$.*
- *$T_r^w \subset T$ denotes the set of tasks to reuse output for checkpoint τ_c^w.*
- *$\mathrm{MCR}(x, y)$, is a mode-change request function to switch from mode x to y such that $x, y \leq n$.*

To ensure data consistency and prevent data corruption, our work assumes the presence of distributed consensus protocols [20]. In a synchronous mode-change protocol, distributed consensus problems will not occur because the system allows the tasks of the new mode to execute only after the tasks of the old mode. On the other hand, to avoid inconsistency in an asynchronous mode-change protocol, any task should not access updated data that are incomplete due to preemption.

IV. Using Checkpoints in Multi-mode Systems

Using checkpoints in real-time systems is different than that employed in a database system [15]. A checkpoint in a multi-mode system stores the saved state of tasks that have completed execution or the tasks that have completed after the last checkpoint. The saved state contains everything that is necessary to continue from the checkpoint.

Example 1. *Consider a scheduling model $R = (W(M_1, M_2), RM)$ that has two modes in the workload W. Mode M_1 has three tasks: $\tau_1(5, 5, 1)$, $\tau_2(9, 9, 5)$, and $\tau_3(22, 22, 3)$. Mode M_2 has also three tasks: $\tau_1(5, 5, 1)$, $\tau_2(9, 9, 5)$, and $\tau_3(24, 24, 1)$. Assume that the model uses a checkpoint τ_c designed as a task which has the specification $(10, 10, 1)$. The modes share the common tasks τ_1 τ_2. Suppose that the system already completed τ_1 and τ_2, and the state is saved at τ_c which is associated with τ_1 and τ_2. If an MCR occurs after τ_2 completes execution to switch to M_2, then the system can use the checkpoint τ_c to retrieve the information of completed τ_1 and τ_2.*

Rollback-supported mode changes in real-time systems impose challenges on where and how many checkpoints to use. This analysis also depends on the overhead associated to checkpoints and requires feasibility analysis. The overhead

analysis involves storing tasks information all that require using checkpoints. Feasibility analysis involves meeting deadlines of all tasks in the presence of MCRs and avoids using checkpoints if tasks do not support them. Our approach for placing checkpoints is based on the RM scheduling policy.

A. Workflow

In this paper, checkpoints are designed using tasks which are intended to be scheduled using a scheduling policy. If periodicity is required for some tasks while changing modes, the offset will be calculated according to the periodic activations. After that schedulability analysis is performed to check the validity of the schedule, and consistency is ensured through using synchronization protocols while running the schedule in an application. The workflow of the approach involves:

(a) specification of checkpoints and tasks. A periodic task can abstract the timing requirements of checkpoints. This timing requirement can be specified at the design time and validated using schedulability analysis. The overhead of a checkpoint involves storing data at checkpoints. Transition to a checkpoint while changing modes and retrieving data are associated with tasks using rollbacks. While executing any of a set of tasks, we assume that the system uses the recent checkpoint that stored the completed tasks from the periodically placed checkpoints. Although transitions from different tasks to a checkpoint upon an MCR can occur, the switching overhead can be bounded by the maximum of all transition overheads because only one transition can be active when changing modes.

(b) choosing a scheduling policy. Scheduling policy determines how the tasks are prioritized for them to execute. Scheduling policies can either consider independent tasks or dependent tasks in the system. Independent tasks can execute in any processors and any order depending on the priority. However, dependent tasks must execute one after another to preserve the precedence relations.

(c) schedulability analysis. Schedulability analysis guarantees whether a particular task set meets the timing requirements. Therefore, if a checkpoint is modelled as a periodic task, the validity and feasibility of using the checkpoint can be checked using utilization, or supply and demand bound functions [18] under a scheduling algorithm. This analysis ensures that the system always meets the deadline.

(d) consistency. In a synchronous protocol, consistency problems will not occur because the new mode tasks are activated after the old mode tasks. In an asynchronous protocol, a task should avoid accessing data that is not updated fully due to preemption. Shared resources must be used in a consistent way to avoid data corruption. Synchronization protocols can be used to ensure the consistency of shared data or to avoid data corruption during the steady state and the transition.

B. Checkpoint-based Minimum Single Offset Protocols

Existing synchronous protocols for both uniprocessor and multiprocessor can be modified to incorporate the capability of using checkpoints. The synchronous minimum single offset protocol can use checkpoints to avoid re-execution of common tasks. When an MCR occurs, the synchronous minimum single offset protocol finishes the execution of the running tasks and disables the tasks that are not running in the old mode before enabling the new mode tasks.

As like synchronous protocols, checkpoints and the associated rollbacks can also be applied to asynchronous protocols for uniprocessor and multiprocessor. The asynchronous minimum single offset protocol schedules the running tasks of the current mode and the tasks in the new mode together. The asynchronous minimum single offset protocol enables the new mode tasks as soon as possible upon an MCR. The priorities of the running tasks of the old mode are assigned according to the scheduling policy, but higher than the new mode tasks during the transition.

C. Mode-change Delay Analysis

A mode-change delay (ϕ) is the time lag that the system experiences during the transition from an old mode to the new mode. Specifically, it is the time between the MCR and the completion of the last task of the old mode. Depending on the mode-change protocol the system uses, it experiences the mode-change delay. Many mode-change protocols exist, but we consider minimum single offset protocol as an example in the system model. Any mode-change protocols and any scheduling algorithm can use the idea of using checkpoints presented in the paper.

This section explains the principles to calculate the mode-change delay for synchronous and asynchronous minimum single offset mode-change protocols and the differences in the mode-change delay calculation between existing approach and our approach. The mode-change delay analysis applies to not only single processor systems but also multiprocessor systems under the assumption of partitioned scheduling in the system model.

In a synchronous mode-change protocol, when an MCR occurs, any task of the old mode that is active must be completed before executing any new mode tasks. Old mode tasks must be completed to preserve data consistency. On a single processor platform, in the worst case, the mode-change delay for the synchronous minimum single offset protocol [16] (Equation 1) is (such that $\phi'_n = \phi'_{n-1}$) the summation of the worst-case execution time of tasks of the old mode that are active ($\tau_i \in T_o$) and the successive releases of the common tasks ($\tau_j \in T_u$). All tasks of the new mode must wait until the tasks of the old mode get executed. In an asynchronous mode-change protocol, the new mode tasks can be scheduled together with the old mode task after a mode-change. The old mode tasks that are ready to run may need to schedule before the new mode tasks in the worst-case because of higher priorities. Therefore, the mode-change delay for the asynchronous minimum single offset protocol remains the same as in Equation 1 in the worst-case.

$$\phi'_n = \sum_{\tau_i \in T_o} e_i + \sum_{\tau_j \in T_u} \left\lceil \frac{\phi'_{n-1}}{p_j} \right\rceil e_j \tag{1}$$

In a checkpoint-based protocol, the common tasks can be avoided for re-execution. Therefore, the worst-case mode-

change delay ϕ for a checkpoint-based minimum single offset protocol is (such that $\phi_n = \phi_{n-1}$),

$$\phi_n = \sum_{\tau_i \in \{T_o - T_r\}} e_i + \sum_{\tau_j \in T_u} \left\lceil \frac{\phi_{n-1}}{p_j} \right\rceil e_j \tag{2}$$

If the system has no tasks to rollback, an MCR will cause the new mode to start from the beginning without reusing any previous execution. This results in the same mode-change delay for systems with or without checkpoints. If the system has tasks to rollback, then less number of tasks will need to run in the new mode, which eventually will make processing faster for the new mode.

Adding a set of checkpoints as instances of a periodic task to the protocol can avoid executing a set of common tasks between the old and new mode. However, overheads are associated with using checkpoints. A checkpointing overhead (e_c) is associated with storing information on completed common tasks. The checkpointing overhead depends on the number of parameters of the tasks, which is represented as a function of task ($\mathrm{S}(\tau_i)$). The summation of all the parameters represents the overhead of all parameters of all common tasks that are completed before a mode change. This yields,

$$e_c = \begin{cases} \sum\limits_{\tau_i \in T_r^w} \mathrm{S}(\tau_i) & \text{if } \exists \tau_i \text{ for } \mathrm{MCR}(x, y) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

A transition overhead (e_t) is incurred if the system uses a checkpoint for mode changes. The transition overhead depends on the number of successfully completed tasks stored at checkpoints. If an $\mathrm{MCR}(x, y)$ occurs from mode x to mode y, and a checkpoint τ_c is used to retrieve all the successfully executed common tasks between x and y. The overhead depends on the number of stored parameters of the tasks, which is represented as a function of task ($\mathrm{U}(\tau_i)$). The overhead represents the time required to retrieve all stored parameters of common tasks that can be reused when the mode change occurs. This yields,

$$e_t = \begin{cases} \sum\limits_{\tau_i \in T_r^w} \mathrm{U}(\tau_i) & \text{if } \exists \tau_i \text{ for } \mathrm{MCR}(x, y) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Therefore, the transition overhead e_t can either be zero or the time required to retrieve the maximum number of completed tasks executions at checkpoint τ_c^w (Equation 4). A checkpoint-based mode-change is efficient because the checkpointing overhead and the transition overhead are negligible because of avoiding re-execution of common tasks. If the system has no common tasks, the overheads are also tolerable due to not using the checkpoints.

Continuing Example 1. On a single processor platform without using checkpoints, the worst-case mode-change delay for synchronous or asynchronous protocols using RM is $\phi'_n = \phi'_{n-1} = 10$. However, on a single processor platform, using the checkpoint τ_c associated to τ_1 and τ_2, the worst-case mode-change delay is reduced to $\phi'_n = \phi'_{n-1} \leq 10$, if $0 \leq e_t \leq 7$. Since the example only has few tasks and variables, it is quite likely that $e_t < 7$ will hold. Our detailed analysis on practical overhead in LITMUS$^{\mathrm{RT}}$ supports this intuition.

D. Checkpoints Bound Analysis

Schedulibility analysis validates the specifications of tasks and checkpoints. If the schedulability analysis fails, the system designers can adjust the specification for placing checkpoints. The system can operate only in one mode at any time. Therefore, the utilization of the workload under the RM scheduling is:

$$\max_{M_k \in W} \Big(\sum_{\tau_i \in M_k} \frac{e_i}{p_i} \Big) \leq n(2^n - 1) \tag{5}$$

The bound on the number of checkpoints varies on the length of time interval. Therefore, given that a system has periodic tasks and places checkpoints periodically with an execution time e_c, Equation 6 determines the checkpointing period p_c such that $p_c > 0$. Equation 6 determines a bound on the number of checkpoints for a maximum of δ mode changes with a worst-case mode-change delay ϕ for a maximum of n number of tasks where $H = \mathrm{LCM}(p_1, \ldots, p_n, p_c)$ is the hyperperiod.

$$\max_{M_k \in W} \Big(\sum_{\tau_i \in M_k} \frac{e_i}{p_i} \Big) + \frac{\delta\phi}{H} + \frac{e_c}{p_c} \leq n(2^n - 1)$$

$$\therefore p_c \geq \frac{1}{n(2^n - 1)} - \frac{1}{\max_{M_k \in W} \big(\sum\limits_{\tau_i \in M_k} \frac{e_i}{p_i} \big)} - \frac{H}{\delta\phi} \tag{6}$$

V. Experimental Analysis

The experimental analysis section of the paper discusses the overhead to use checkpoints in a real-time operating system such as LITMUS$^{\mathrm{RT}}$ under RM scheduling policy. Using an example in-vehicle multi-mode system (IVM) case-study, it is shown that the overhead for using checkpoints is low compared to the advantages in multi-mode systems. Using the experiments performed in LITMUS$^{\mathrm{RT}}$, an approximation of checkpoint execution time is made, and this is used to calculate checkpointing period for a different number of MCRs. Finally, simulations on the example case-study and randomly generated tasks [1] for different input parameters demonstrate that using checkpoints reduces the mode-change delay and the total execution time of the new mode when common tasks exist in the system.

An in-vehicle infotainment system is used in automobiles, or other forms of transportation, to provide audio and visual entertainment as well as navigation. The architecture of an in-vehicle infotainment system has repeatedly been designed over the last few years because of the demand for eco-friendly cars that have all the latest facilities. Therefore, we envision an IVM architecture (Fig. 1) that combines multiple subsystems such as climate control and navigation. These subsystems correspond to different modes with some common tasks (e.g., initialization). The envisioned IVM architecture contains different independent tasks and a mechanism to schedule them deterministically and correctly to deliver the right output at the right time. Since the proposed scheme of using rollback is application independent and works solely based on task

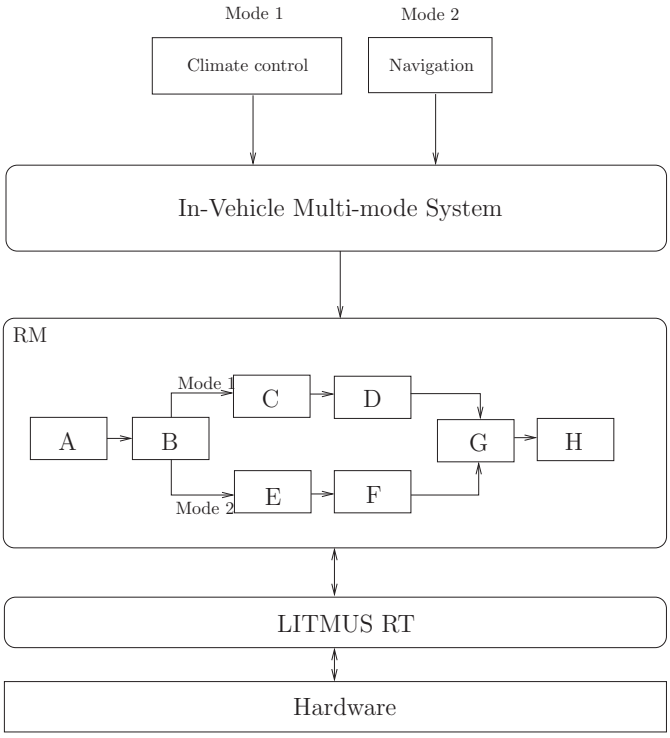

Fig. 1. A reference architecture for an in-vehicle infotainment system

models as defined in Section III, an engineer may change the system-of-interest from the envisioned IVM.

To demonstrate that the overhead of using the proposed scheme is low, we have implemented the proposed approach in LITMUS^{RT} [7], a real-time extension of the Linux kernel, in a dual-core x86 machine with 6GB DDR3 memory. The experimental results on LITMUS^{RT} provides analysis on the overhead of placing checkpoints and using them.

To generate and run implementation according to the high-level requirements, the specifications are converted into tasks that belong to multiple modes. Since several research works exist [3] on how to specify timing requirements for real-time systems, this work avoids discussion on how to get specifications of tasks. Therefore, in this work, we consider that the IVM has some independent tasks classified into modes. An example workload consisting of six tasks in both modes exists as shown in Fig. 1. Table I shows the specifications of tasks that are characterized by the timing specification (i.e., period, deadline, and execution time) and the modes that they belong. Modes have four common tasks: A, B, G, and H.

The tasks of multi-mode systems considered in the experimentation are scheduled using partitioned RM. For mode changes, MCR occurs randomly. We do not allow new tasks to be admitted into the system. The goal of the experimentation is to assess the proposed system quantitatively using the following metrics:

- Checkpoint overhead (e_c): This metric measures the time required to store state variables when saving a checkpoint.
- Transition overhead (e_t): This metric measures the time required to execute a checkpoint upon a mode change.
- Checkpoint period (p_c): This metric measures how frequently the system uses checkpoints.
- Mode-change delay (ϕ): This metric measures the delay during changing modes.

TABLE I
TASKS SPECIFICATIONS(IN MICROSECONDS)

Tasks	Mode
A(10,10,1)	1,2
B(15,15,2)	1,2
C(20,20,1)	1
D(25,25,2)	1
E(20,20,2)	2
F(25,25,1)	2
G(30,30,2)	1,2
H(35,35,2)	1,2

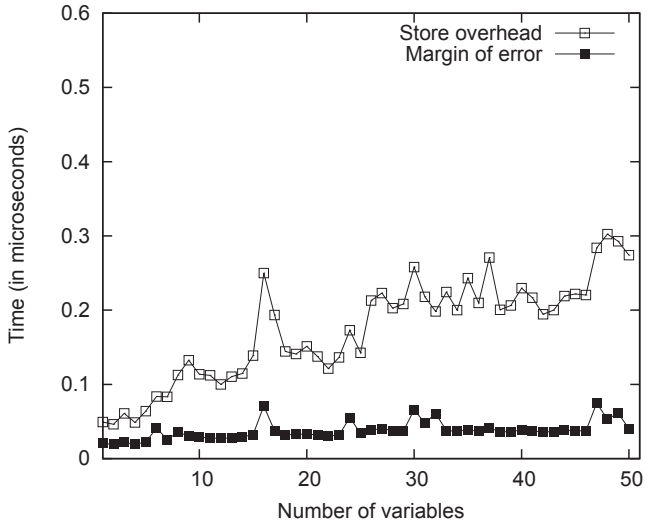

Fig. 2. Analysis on overhead due to storing variables

Checkpoint overhead. Checkpoint overhead is proportional to the number of variables (each of 2 bytes) stored or written in the memory. With an increasing number of variables written in the memory, Fig.2 shows that the checkpoint overhead is low with the 95% confidence level, devised from the standard deviation and margin of error.

Transition overhead. Transition overhead is the sum of the delay to move to a checkpoint upon an MCR and the overhead to retrieve state variables. We define the delay to move to a checkpoint upon an MCR as transition latency. To measure the transition latency, some trials prompt the system to execute checkpoints at different points in time, and record the delay for each of them for ten variables, each of 2 bytes. Table II shows the transition latency with an interval of 95% confidence. The worst-case transition latency is the maximum of transition latency of all tasks. The overhead to retrieve variables is low akin the checkpoint overhead (Fig. 3).

Mode-change delay. Mode-change delay analysis explains the time required to switch to a new mode when an MCR oc-

TABLE II
TRANSITION LATENCY ANALYSIS(IN MICROSECONDS)

Tasks	Latency	StdDev	CI(95%)
A	1.4945	0.7177	[1.4262-1.5636]
B	1.6682	0.4719	[1.6198-1.7106]
C	1.4450	0.8639	[1.3622-1.5278]
D	1.5855	0.6945	[1.5162-1.6548]
E	1.5490	1.6845	[1.4028-1.6952]
F	1.2687	0.4438	[1.2245-1.3129]
G	1.4849	0.5378	[1.436-1.5338]
H	1.2466	0.4573	[1.2038-1.2894]

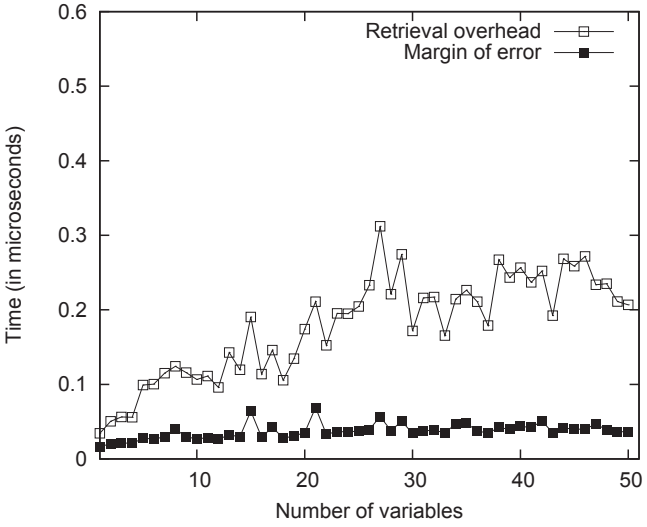

Fig. 3. Analysis on overhead due to retrieving variables

curs. Mode-change delay differs among existing and proposed methods for the case-study because it has some common tasks. Considering the MCR for the worst-case, we get $\phi' = 10\mu s$ and $\phi = 5\mu s$ if the transition overhead $e_t = 3\mu s$. The reduction in mode-change delay will be higher for using rollback if the execution time of common tasks is greater than that specified in the case-study.

To analyze further on the mode-change delay, we use Bini's random task generator [1] to generate tasks with a 25% probability of having common tasks in different modes. In the simulations, we vary the probability from 10% to 50% of these tasks as common tasks. However, no common task may present in the system when the number of tasks in the system is low. Simulation results show the differences in mode-change delays for the existing minimum single-offset protocol, and the proposed enhancement. For simplicity, overhead, which is demonstrated already as low, is not considered in these simulations. We analyze worst-case mode-change delay, (1) for a variable number of modes but a constant number of mode changes and tasks in each mode (Fig. 4), (2) for a variable number of tasks but a constant number of mode changes and modes (Fig. 5), and (3) for a variable number of repeated mode changes but a constant number of modes and tasks (Fig. 6).

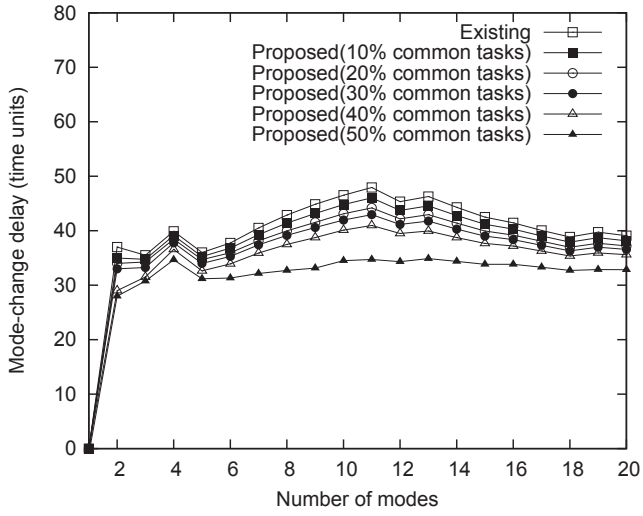

Fig. 4. Analysis on variable number of modes for 10 tasks and uniform MCRs

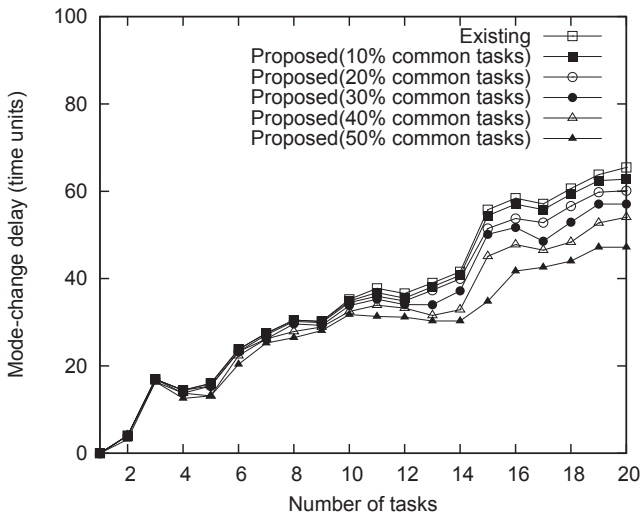

Fig. 5. Analysis on variable number of tasks for 5 modes and uniform MCRs

Simulation study uses generated random tasks to demonstrate that the proposed enhancement to use rollback to reduce mode-change delays will be significant if common tasks exist in the system. Fig. 4 shows that while varying a number of modes, the mode-change delay is reduced for increasing number of common tasks. Since the tasks are generated randomly, the percentage improvement of the proposed scheme over existing approach is not exact, because mode-change delay for both schemes is the same in the absence of common tasks. Fig. 5 shows that while varying a number of tasks in each mode, the mode-change delay is reduced for increasing number of common tasks. Fig. 6 shows while varying repeated number of MCRs, the mode-change delay is reduced for increasing number of common tasks.

Throughput analysis. Throughput analysis explains the ef-

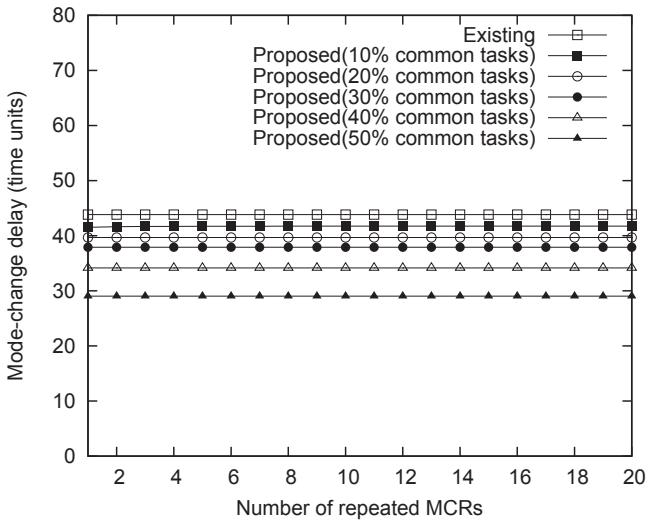

Fig. 6. Analysis on variable number of mode changes for 10 tasks and 5 modes

ficiency of the system regarding utilizing system resources. To demonstrate the advantages of using checkpoints, throughput L for two modes (i.e., old and new mode) is calculated as the following:

$$L = \sum_{\tau_i \in T_o} \frac{e_i}{p_i} + \sum_{\tau_j \in \{T_w - T_u\}} \frac{e_j}{p_j} + \frac{e_c}{p_c}$$

Throughput increases between two modes because of using checkpoints to avoid re-executing common tasks. A repeated MCR indicates a similar mode change request as previous between two modes. A checkpoint-based multi-mode system can avoid more re-executions of common tasks if the number of repeated MCR increases. This also increases the throughput of the system.

VI. Conclusion

Checkpoints have been extensively used in the area of fault-tolerance in different application domains including real-time systems. However, checkpoints are also useful in multi-mode systems, in addition to providing fault-tolerance. This work has demonstrated the inclusion of checkpoints in a multi-mode system to utilize the output of prior successful completed executions of common tasks when changing to a new mode. This increases the overall throughput of a system because of reducing the number of task executions. This is demonstrated by running examples on LITMUS^{RT}. Therefore, the inclusion of checkpoints in a multi-mode system is useful for safety-critical systems where reliability and efficiency matter.

References

[1] Random Task Generator for EDF. www.retis.sssup.it/~bini/resources/c/. Visited October. 2013.

[2] A. Azim, S. Sundaram, and S. Fischmeister. An Efficient Periodic Resource Supply Model for Workloads with Transient Overloads. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[3] I. Bate, P. Nightingale, and A. Cervin. Establishing Timing Requirements and Control Attributes for Control Loops in Real-time Systems. In *Euromicro Conference on Real-Time Systems*, pages 121–128, 2003.

[4] A. Bertossi, M. Bonometto, and L. Mancini. Increasing Processor Utilization in Hard-Real-Time Systems with Checkpoints. *Real-Time Systems*, 9(1):5–29, 1995.

[5] N.S Bowen and D.K Pradham. Processor and Memory-based Checkpoint and Rollback Recovery. *Computer*, 26(2):22–31, 1993.

[6] V. Brocal, P. Balbastre, and R. Ballester. Task Period Selection to Minimize Hyperperiod. In *Emerging Technologies & Factory Automation (ETFA)*, 2011.

[7] J. M. Calandrino, H. Leontye, A. Block, U. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, 2006.

[8] N. Chen and S. Ren. Adaptive Optimal Checkpoint Interval and Its Impact on System's Overall Quality in Soft Real-Time Applications. In *ACM Symposium on Applied Computing*, pages 1015–1020, 2009.

[9] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[10] K. Lee and J. Eidson. IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *In 34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 98–105, 2002.

[11] V. Nelis, J. Goossens, and B. Andersson. Two Protocols for Scheduling Multi-mode Real-Time Systems upon Identical Multiprocessor Platforms. In *21st Euromicro Conference on Real-Time Systems*, pages 151–160, 2009.

[12] L.T.X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 91–100, April 2011.

[13] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):389–402, March 2009.

[14] Dumitru Potop-Butucaru, Akramul Azim, and Sebastian Fischmeister. Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures. In *Proc. of the International Conference on Embedded Software (EMSOFT)*, Scottsdale, Arizona, USA, October 2010.

[15] S. Punnekkat, A. Burns, and R. Davis. Analysis of Checkpointing for Real-Time Systems. *Real-Time Systems*, 20(1):83–102, 2001.

[16] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26:161–197, 2004.

[17] S. Samii, P. Eles, P. Zebo, and A. Cervin. Quality-driven synthesis of embedded multi-mode control systems. In *Design Automation Conference, 2009. DAC '09*, pages 864 –869, july 2009.

[18] I. Shin and I. Lee. Compositional real-time scheduling framework. In *IEEE Real-Time Systems Symposium*, pages 57–67, 2004.

[19] I. shin and I. Lee. Periodic Resource Model for Compositional Real-time Guarantees. In *Technical Report*, pages 21–15, 2010.

[20] A. Tanenbaum and M. Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002.

[21] K. Tindell and A. Alonso. A Very Simple Protocol for Mode Changes in Priority Preemptive Systems. Technical Report, Universidad Politcnica de Madrid, 2002.

[22] P.M. Yomsi, V. Nelis, and J. Goossens. Scheduling multi-mode real-time systems upon uniform multiprocessor platforms. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1 –8, Sept. 2010.

[23] M. Zhang, J. Shi S. Shen, and T. Zhang. Simple Clock Synchronization for Distributed Real-Time Systems. *IEEE International Conference on Industrial Technology*, pages 1–5, 2008.

[24] Y. Zhang and K. Chakrabarty. Fault Recovery Based on Checkpointing for Hard Real-Time Embedded Systems. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 320–328, 2003.

Acknowledgements.

This research was supported in part by the University of Ontario Institute of Technology (UOIT) startup grant (154052-3130).