Virtual Single-Core

for Multicore Real-Time Computing

Lui Sha, Fellow, Jung-eun Kim, Member, Jose Meseguer, Member, Peter C. Ölveczky

***Abstract*—** This paper introduces *Virtual Single-Core (VSC)* technology that allows engineers to use a group of cores in a multicore computer as if the group of cores were a larger single-core computer.

Multicore technology has many benefits, such as increased CPU bandwidth per chip. However, when used as is, inter-core interferences can be severe. Because of the potential for large and random delay spikes, the U.S. Federal Aviation Administration (FAA), European Aviation Safety Agency (EASA), and Transport Canada specify that only *one* core can be used, unless inter-core interference is specifically defined and handled. In addition, DO-178C: Software Considerations in Airborne Systems and Equipment Certification is for single-core chips only.

Single-core Equivalence (SCE) technology partitions the resources shared by cores in such a way that each core can be used as if it were a single-core computer. SCE is an effective solution that address certification authorities' concerns of intercore interference. However, a core in a multicore chip is often slower than a fast single-core chip. Therefore, a large multi-thread (task) application may not be scheduled within a core.

Virtual Single-Core (VSC) technology extends the SCE technology so that a group of cores can be used to schedule a large application as if the VSC were a larger single-core computer. VSC greatly facilities the migration of certified avionics software from single-core computers to multicore computers.

***Index Terms*— Multicore Real-Time Computing, Avionics, Avionics software migration, Certification, Virtual Single Core.**

1.0 Introduction

Multicore technology has many benefits, such as increased CPU bandwidth per chip. However, when used as is, inter-core interferences can be severe. For example, measurements provided by Lockheed Martin Space System Integration Lab testbed showed that using an 8 core (Freescale P4080) chip, a task's WCET can increase by as much as *600*% when it runs concurrently with logically independent tasks in other cores. Such worst-case occurred when7 out of 8 cores were used, not when all 8 cores are used [2]. Because of the potential for large and random delay spikes, the U.S. Federal Aviation Administration (FAA), European Aviation Safety Agency (EASA), and Transport Canada specify that only *one* core can be used unless inter-core interference is specifically defined and handled [17]. In addition, DO-178C: Software Considerations in Airborne Systems and Equipment Certification has been developed for single-core chips [17]. Under SCE's partition and isolation mechanisms, the inter-core interferences are tightly bounded and accounted for so that each core can be used as if it were a single-core computer [2].

However, a core in a multicore chip often runs slower than a fast single-core computer. Some big applications are already not able to fit into a core in a multicore chip. Future applications may even be bigger. This paper extends the SCE approach by creating a new technology called Virtual Single-core (VSC). VSC technology enables users to reuse single-core real-time scheduling technology, Generalized Rate Monotonic Scheduling (GRMS), which has been approved by D.O. 1778B/C in safety-critical avionics software.

From a schedulability analysis perspective, when a big application's tasks cannot be fit into a core, we must distribute the tasks into different cores. However, tasks distributed to different cores may share variables. To support the reuse of GRMS' priority ceiling protocol (PCP) as is, all the critical sections of tasks with shared variables must be executed within a single-core to ensue the PCP's properties: i) freedom from dead locks; and ii) a higher priority task can be blocked by lower priority tasks at most once [3]. The core used to execute all the critical sections of tasks within a VSC is called synchronization cores. Other cores of VSC are called execution cores.

This work was supported in part by Office of Naval Research N00014-17-1-2783, by National Science Foundation CNS 1302563, The views expressed here are those of the authors and do not represent the views of the sponsors.

When a task assigned to an execution core needs to access a shared variable, this task makes a lock mutex call to the RTOS, RTOS moves this task to the synchronization core and synchronize the locking using PCP protocol. Once this task exits its critical section, the RTOS moves it back to its assigned execution core. We call such a task as a multicore task because it executes on both its assigned execution core and the synchronization core.

This paper's contribution to schedulability analysis is a task transformation algorithm that maps a set of *multicore tasks L* to a task set *L consisting of *only single-core tasks*. *L can be analyzed by traditional single-core schedulability analysis. If a task in *L is schedulable, then this in L is also schedulable in the VSC under GRMS (See Section 2.4).

1.1 Related Work

Real-time multicore computing is an active area of research. For example, Anderson et al. investigated a cache-aware Pfair-based scheduling scheme for real-time tasks on multicore platforms [10]. Pellizzoni et al. investigated how to reduce memory interferences [11]. Rajkumar was the first in the development of multiprocessor synchronization protocol [9], and Brandenburg et al. did a comparison study of multiprocessor real-time synchronization protocols M-PCP, D-PCP, and FMLP[8].

A particular approach to real-time multicore computing is known as Single-core Equivalent (SCE) technology, which partitions the resources shared by cores in such a way that each core can be used as if it were a single-core chip [2]. This is a set of co-designed protocols that include Yun et. al.'s work on DRAM management [12] and on memory bus bandwidth management [13]. Kim et al. studied the I/O in the context of Integrated Modular Avionics [14], Mancuso et al. investigated real-time cache management [16], and the estimation of WCET in a multicore chip [15].

However, none of the above addresses the big application challenge in a way that can be analyzed by scheduling method approved for safety-critical avionic applications.

2.0 Virtual Single-core

VSC builds on Single-core Equivalent (SCE) technology; that is, all the resources shared by cores are partitioned accordingly.

In SCE, *WCET(k)* means that only *k* cores in an *m* core multicore chip are used. If we use *2* cores and disable *6* cores in an 8 core chip, we have *WCET(2)* for a task. A task's *WCET(m)* can be measured or computed from *WCET(1)* [15]. In an 8 core chip, a task's *WCET(8)* is just its traditional *WCET* in a core with *1/8* of the shared resources since SCE's default is to evenly partition.

2.1 Assumptions and Definitions

When all the tasks of a multi-thread (task) application cannot be scheduled (fit) within a single-core of a multicore chip, it is said to be a "big-application" for that chip.

Definition 1: Synchronization tasks: Tasks using shared variables are referred to as synchronization tasks in this paper. Otherwise, they are non-synchronization tasks. From a real-time scheduling perspective, non-synchronization tasks can be scheduled independently. Thus, we call them independent tasks in schedulability analysis.

Assumption 1: **VSC core allocation:** Since *WCET(m)* under SCE is just the traditional *WCET* in a core with *1/m* of the shared resources, we will use the term *WCET* instead of *WCET(m)* in the rest of this paper.

Remark: Since aperiodic events can be handled by periodic servers, we only consider periodic tasks in this paper.

Definition 2: Task Model

- Each task T_i has period P_i and *WCET* e_i, with its deadline at the end of the period.
- Task T_i may be decomposed into *3* disjoint code segments, consisting of alternating non-critical section and critical section code segments, i.e., T_i:{$e_{i,1}$, $cs_{i,2}$, $e_{i,3}$}, where $e_{i,1}$ and $e_{i,3}$ denote non-critical code segment and $cs_{i,2}$ denotes a critical section code segment.
- Tasks are indexed so that a lower number indicates a higher priority.

Remark: It is possible to generalize the job model into a series of alternative non-critical and critical section code segments.

Definition 3: *syn_core* and *exe_core*: To make a big-application schedulable, we will keep all the shared variables and critical section code in a dedicated core in VSC. We call this core as *syn_core*. All other cores in this VSC are said to be *exe_cores*.

In a typical real-time system, the length of critical sections is only a small fraction of execution time. In addition, while a multi-threaded big application cannot fit all its tasks within a core, each task (thread) by itself has utilization no more than 100%.

Assumption 2: **Max critical sections and thread utilizations**: A big-application's critical sections, by themselves only, are schedulable in the *syn_core*. The utilization of any task in a core is less than or equal to 100%.

Assumption 3: Generalized Rate Monotonic Scheduling (GRMS) is used to schedule tasks: In each VSC core, we use generalized rate monotonic algorithm (GRMS) [5] to schedule tasks

Definition 4: **VSC Task Synchronization**: All tasks in a VSC execute their critical sections in the synchronization core. The Priority Ceiling Protocol (PCP) lock protocol [3] is used to synchronize the executions of all the critical sections of a VSC's tasks in the synchronization core.

Definition 5: Virtual Single-core (VSC): in a multicore computer, a set of cores is said to form a VSC, if:

- SCE is used to partition the shared resources.
- GRMS developed for single-core computer can be used as is to determine if each task is schedulable, even if the task has to use more than one core for its execution.

Definition 6: **Single-core and multicore tasks:** A single-core task is one that executes all its code segments within a core. Otherwise, it is a multicore task. A single core task can reside in the synchronization core as long as it is schedulable.

Definition 7: Multicore task execution model: A multicore task starts and ends in its assigned *exe_core*(s). When a multicore task makes a lock *mutex* call, the RTOS executes its critical section code segment in the *syn_core*. Once a multicore task finishes its critical section and unlocks all the *mutexes*, RTOS executes its next non-critical section code in its assigned *exe_core*.

Definition 8: VSC core group configuration: We will define a simple task allocation heuristic to fit a big application's tasks into a small number of cores. The resulting number of cores that can schedule this big application's tasks constitutes a VSC. If there is remaining capacity in a VSC, we can use it for small applications that can be fit into a core with spare capacity.

Remark: The (near) optimal task allocation algorithm for VSC is outside the scope of this paper.

2.2 VSC task allocation rule for a big-application

Phase 1: Initialization

Step 1: We allocate all the tasks of big-application ***A*** to core C_s, the *syn_core*. The shared variables are statically allocated in the memory space of C_s.

By assumption, big-application ***A*** is too big to be schedulable (fit) in one core, and we must move some of its tasks from core C_s to execution core(s).

Phase 2: Making an unschedulable task schedulable in the synchronization core.

Let C_1 be the *syn_core*. That is, $C_{s=1.}$

Step 2.1: **Identify the first task that is unschedulable**[1]: Perform schedulability analysis starting with the highest priority task first and going down the priority order. Stop at the first unschedulable task. Let this unschedulable task be T_j.

Step 2.2: Moving higher priority independent tasks to an *exe_core*: We move independent tasks with priorities higher than task T_j one by one, from *syn_core* $C_{s=1}$ to *exe_core* C_2. There are two cases:

- **Case 1**: Task T_j becomes schedulable after some or all independent tasks are moved to C_2. Jump to Step 3.1.
- **Case 2**: Task T_j still is unschedulable. Continue to Step 2.3.

Step 2.3: **Moving higher priority synchronization tasks to *exe_core***: Starting from with higher priority synchronization tasks first, we move them one by one to execution core C_2, until T_j becomes schedulable.

Phase 3: Making all tasks in the synchronization core schedulable

Step 3: If there is any remaining task that is still unschedulable, use the Phase-2 steps to make it schedulable. Repeat this process until all the tasks in the synchronization core become schedulable.

Remark: By Assumption 2 on the total utilization of critical sections, Step 3 can always be accomplished.

Phase 4: Making tasks schedulable in execution cores:

Step 4.1: If a task T_j in core C_2 is not schedulable, we move tasks with higher priority than task T_j to core C_3 until task T_j is schedulable. We repeat this procedure until all tasks in C_2 are schedulable

Step 4.2: We repeat this process in step 4.1 in each execution core until all the tasks are schedulable.

[1] See Section 2.4 on schedulability analysis.

At the end of phase 4, all the tasks of big-application ***A*** become schedulable. The number of cores in use is that of those cores used by the VSC for big-application ***A***.

2.3 VSC Task Synchronization

Theorem 1: In a VSC, each task's critical section is deadlock-free and can be blocked at most once.

Proof: By Definition 7 on the execution model, all

the critical section code segments will be executed in the *syn_core*. By Definition 4, critical sections in the *syn_core* are scheduled by the (single-core) PCP. By the property of PCP [3], the theorem follows. QED.

Let the worst-case overhead to move the execution of a task from core C_i to core C_j be $H_{i,j}$. Let the worst-case overhead to move a task back from core C_j to core C_i be $H_{j,i.}$ For simplicity, we will let $H = \max(H_{i,j}\ ,\ H_{j,i})$.

Theorem 2: **Max task moving overhead**: The worst-case task moving overhead is *2H* at execution core C_e, and *2H* at synchronization Core C_s.

Proof: When a multicore task is moved between *exe_core* and *syn_core*, it creates overheads. At the *exe_core*, the overhead of moving a task to and then back from *syn_core* is bounded by *H+H = 2H*. QED.

2.4 Reuse Single-Core Exact Schedulability Analysis

In this section, we develop the technology that enables the reuse of the widely used (single-core) exact schedulability analysis by transforming of the given set of tasks **L** into a new set of tasks ***L** such that:

- We can analyze the task schedulability in *L using a (single-core) exact schedulability analysis as is.
- If the tasks in ***L** are schedulable, the tasks in **L** are also schedulable.

2.4.1 Multicore Tasks Preemption

So far, we have analyzed the blocking from lower priority tasks. We now analyze the preemption effect of multicore tasks. We begin with an example.

Example 1: As shown in Figure 1, we have one multicore task and two single-core tasks. All the execution times are *WCET*s.

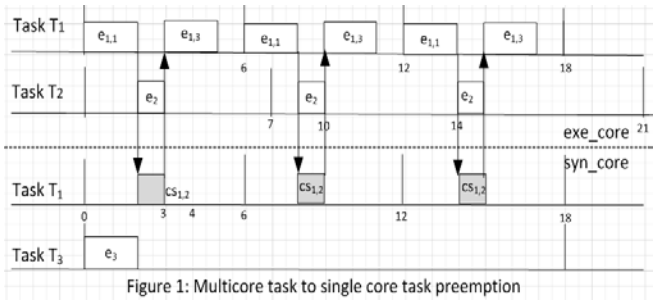

Figure 1: Multicore task to single core task preemption

- Task T_1 is a multicore task with non-critical section code $e_{1,1} = 2$ and $e_{1,3} = 2$ running in *exe_core* and critical section code $cs_{1,2} = 1$ running in the *syn_core*. In addition, its period $P_1 = 6$.
- Tasks T_2 executes only in *exe_core*, and $P_2 = 7$; $e_2 = 1$.
- Tasks T_3 executes only in *syn_core*. In addition, $P_3 = 18$; $e_3 = 2$.

As shown in Figure 1, when task T_1 executes its critical section $cs_{1,2}$ in *syn_core* from $t = 2$ to *3*, task T_2 completes its execution of e_2 at $t = 3$ in the *exe_core*.

The actual execution time can be data-dependent. In single-core scheduling, the reduction of the execution time of a higher priority task cannot increase lower priority tasks response time.

The reduction of the execution time of a multicore task in one core may, however, increase the response time of a lower priority task in another core. Suppose that task T_2 skips the execution of $cs_{1,2}$ under some input data. In this case, $e_{1,1} = 2$ and $e_{1,3} = 2$ will be executed consecutively.

As a result, T_2 is preempted $2+2 = 4$ units and will finish at $t = 5$ instead of at $t = 3$. From the perspective of task scheduling in *exe_core*, executing $e_{1,1} = 2$ and $e_{1,3} = 2$ *without suspension* satisfies the Liu and Layland's critical instant theorem [18].

Also shown in Figure 1, task T_3 executes its e_3 from $t = 0$ to $t = 2$ in the *syn_core* with period $P_3 = 18$. Suppose that the execution of $e_{1,1}$ is also input dependent, and $e_{1,1}$ was skipped. In this case, $cs_{1,2}$ would execute from $t = 0$ *to* $t = 1$. Hence, e_3 will finish in $t = 3$ instead of $t = 2$.

Theorem 3 and Corollary 3 analyze the duration of a multicore task's preemption to a lower priority task, when the execution time of the multicore task changes.

Theorem 3: If we replace a multicore task T_i *:* $\{e_{i,1}, cs_{i,2}, e_{i,3}\}$ with a corresponding single-core task $*T_i$ *:* $\{e_{i,1}, e_{i,3}\}$ in *exe_core*, the preemption to lower priority tasks in *exe_core* is greater or equal to that of T_i.

Proof: When task T_i executes its $cs_{i,2}$ *syn_core*, it suspends its execution in *exe_core* before executing $e_{1,3}$. By removing $cs_{i,2}$, T_i will preempt lower priority tasks with $e_{1,1}$ and $e_{1,3}$ consecutively. By the critical instant theorem, Theorem 3 follows. QED.

Corollary 3: If we replace a multicore task T_i *:* $\{e_{i,1}, cs_{i,2}, e_{i,3}\}$ with a single-core task $*T_i$*:* $\{cs_{i,2}, e_{i,3}\}$. $*T_i$'s preemption to lower priority tasks in *syn_core* is greater

or equal to that of T_i.

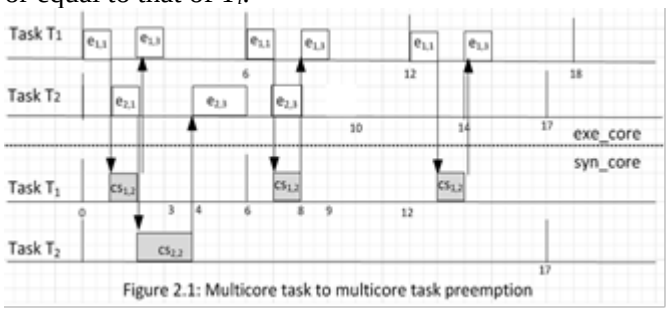

Figure 2.1: Multicore task to multicore task preemption

Example 2: This example focuses on the preemption between two multicore tasks in the *syn_core*. As illustrated in Figure 2.1:

- T_1 has non-critical section code segments $e_{1,1} = 1$ and $e_{1,3} = 1$ running in *exe_core*, and has critical section code segment $cs_{1,2} = 1$ running in the *syn_core*. T_1's period $P_1 = 6$.
- Tasks T_2 has non-critical section code $e_{2,1} = 1$ and $e_{2,3} = 3$ running in *exe_core*, and $cs_{2,2} = 2$ running in the *syn_core*. T_2's period $P_2 = 14$.

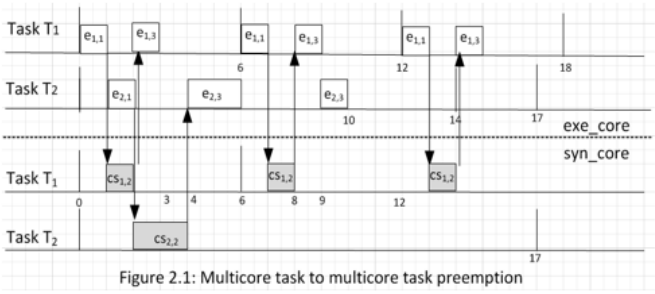

Figure 2.1: Multicore task to multicore task preemption

In this paper, we are interested in finding an easy-to-compute upper bound of the response time of $cs_{2,2}$, $R(cs_{2,2})$.

Notation: The worst-case response time to complete the execution of a critical section $cs_{i,j}$ is denoted as $R(cs_{i,j})$.

Theorem 4: Under the task allocation rule in Section 2.2, a multicore's critical section code segment cannot be preempted by non-critical section code segments.

Proof: Suppose that a multicore task T_j's critical section code segment $CS_{j,2}$ were preempted by a non-critical code segment $e_{i,k}$ in *sync_core*. It follows that $e_{i,k}$ must have higher priority. There are two cases.

Case 1: Suppose that $e_{i,k}$ belongs a multicore task T_i. However, this contradicts Definition 7 on a multicore task execution model, where a multicore task executes all its non-critical section code segment in *exe_core*.

Case 2: Suppose that $e_{i,k}$ belongs a single-core task T_i. However, this contradicts VSC task allocation rule. Under this rule, higher priority single-core tasks must be first moved to *exe_core*, before moving synchronization task T_j to *exe_core* and turning T_j into a multicore task. Theorem 4 follows. QED.

Corollary 4: A multicore task's critical section code segment cannot be preempted by a single-core task.

Proof: Under the VSC task allocation rule, higher priority tasks are moved to *exe_core* until tasks in *syn_core* become schedulable. The corollary follows.

Theorem 4 and its corollary inform us that a multicore's critical section code segment can only be preempted by higher priority multicore tasks' critical session code segments.

Let $\{(cs_{1,2}, P_1), \ldots (cs_{i,2}, P_i) \ldots (cs_{n,2}, P_n)\}$ be multicore tasks' critical section code segments and their periods running in the *syn_core*. Let B_i be the maximal blocking time under PCP to $cs_{i,2}$.

Theorem 5: The maximal response time of $cs_{i,2}$, $R(cs_{i,2})$, can be computed as follows.

$$r^0 = B_i + cs_{1,2} + \ldots + cs_{i,2}$$

$$\ldots$$

$$r^{k+1} = B_i + cs_{i,2} + \sum_{j=1}^{i-1} \left\lceil \frac{r^k}{P_j} \right\rceil * cs_{j,2}$$

If $r^{k+1} = r^k \le P_i$, then $R(cs_{i,2}) = r^k$.

Proof: This is a direct application of the single-core exact schedulability analysis under the (pessimistic) assumption that all the critical section code segments start at the same time. The theorem follows. QED.

Using Theorem 3 and Theorem 5, we can replace the original set of tasks **L**: $\{T_1, T_2\}$ with a new set of tasks ***L**: $\{*T_1, *T_2\}$ such that

- We can analyze each task's schedulability in *L using the (single-core) exact schedulability analysis as is.
- If a task in ***L** are schedulable, then this task in L is also schedulable.

Example 3: We have a set **L** of two tasks T_1 and T_2 :

L: $\{T_1$:$\{e_{1,1}=1, cs_{1,2}=1, e_{1,3}=1\}, T_2$:$\{e_{2,1}=1, cs_{2,2}=2, e_{2,3}=3\}\}$.We want to check if task T_2 is schedulable in the *exe_core*.

- According to Theorem 3, we transform the higher priority task T_1 to $*T_1$ by dropping $cs_{1,2}$ *in* T_1. As a result, the preemption from $*T_1$ *to* T_2 is greater than or equal to the preemption from T_1 *to* T_2.
- According to Theorem 5, we transform the lower priority task T_2 to $*T_2$ by replacing $cs_{2,2}$ *in* T_2 with the

worst-case of the response time of $cs_{2,2}$, $R(cs_{2,2})$. As a result, $*T_2$ is easier to miss its deadline than T_2.

This gives us ***L**: {$*T_1$:{$e_{1,1}=1$, $e_{1,3}=1$}, $*T_2$ ($e_{2,1}=1$, $R(cs_{2,2}) = 3$, $e_{2,3}=3$)}, where $R(cs_{2,2})$ is computed as follows.

- $r^0 = cs_{1,2} + cs_{2,2} = 1 + 2 = 3$
- $r^1 = 2 + \lceil 3 / 6 \rceil * 1 = 3 = R(cs_{2,2})$

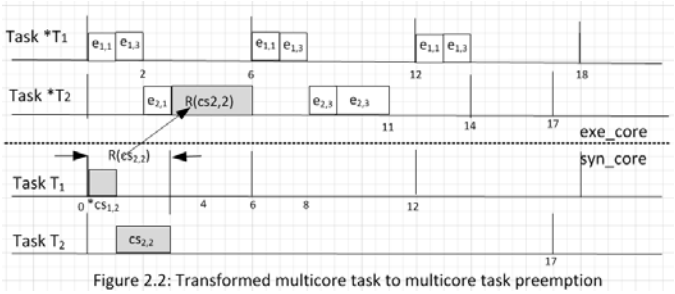

Figure 2.2: Transformed multicore task to multicore task preemption

As illustrated in Figure 2.2., we can now reuse the single-core exact schedulability analysis to determine the upper bound of task T_2's finishing time with the new task set {$*T_1$, $*T_2$}

$r^0 = R(cs_{2,2}) + (e_{1,1} + e_{1,3}) + (e_{2,1} + e_{2,3})$

$= 3 + 2 + 4 = 9$

- $r^1 = 3 + 4 + \lceil 8 / 6 \rceil * 2 = 11$
- $r^2 = 3 + 4 + \lceil 11 / 6 \rceil * 2 = 11$

That is, $*T_2$ completes at $t = 11$. In Figure 2.1, T_2 finishes at $t = 10$, which is bounded by T_2's finishing time at $t = 11$.

The following pseudo-code summarizes the transformation procedure.

Check if a task is schedulable in *exe_core*. Let L_{exe_core} be the given set of m tasks *in exe_core*. As illustrated by Example 3, to check if task T_i is schedulable, we first drop all the higher priority tasks' critical sections and then replace task T_i*'s* critical section $cs_{i,s}$ with its worst-case response time $R(cs_{i,s})$.

Let$*L_{exe_core}(T_i)$ be a transformed task set for analyzing the schedulability of task T_i in *exe_core*.

$*L_{exe\_core}(T_i)$ is initialize to be empty.

{for $j = 1$ to $(i - 1)$ in task set $L_{exe\text{-}core}$

{If (T_j *is a single-core task*)

{*Insert* T_j *to the task set* $* L_{exe\text{-}core}(T_i)$ }

else

{*Insert* $*T_j$ into task set $* L_{exe\text{-}core}(T_i)$ }

}

If Task T_i is a single-core task,

{*insert* T_i into task set $* L_{exe\text{-}core}(T_i)$}

else

{*insert* $*T_i$:($e_{i,1}$, $R(cs_{i,2})$, $e_{i,3}$) into set $* L_{exe\text{-}core}(T_i)$}

Check if a single-core task in *syn_core* is schedulable. In the *syn_core*, we execute all tasks' critical sections and single-core tasks that assigned to the *syn_core*.

Suppose that we have i) n single-core tasks in *syn_core*, and ii) m multicore tasks in the *exe_core*. For each multicore task T_j, we create a periodic task with T_j*'s* period and with execution time equal to T_j*'s* critical section in *syn_core*.

In summary, the transformed task set in the *syn_core* consists of i) the n single-core tasks and ii) a set of periodic tasks with execution times and periods correspond to the critical sections and periods of m multicore tasks respectively. To check if a single-core task is schedulable, we simply use the exact schedulability analysis [4].

Remark: A multicore task's critical section will not start its execution at time $t = 0$ at the *syn_core*. By replacing the execution of m multicore tasks' critical sections with m periodic tasks, all m multicore tasks' critical sections can now start to execute at $t = 0$. According to the critical instant theorem, such transformation maximizes the preemption effect in the execution of these m critical sections.

Example 4: As illustrated in Figure 3, this example illustrates the task allocation and schedulability analysis

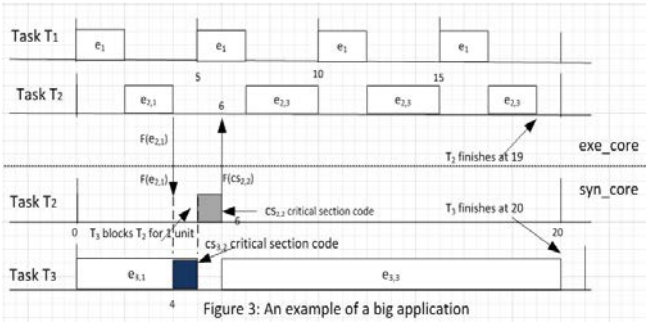

Figure 3: An example of a big application

of a big application.

For simplicity, we assume that the migration overhead has already been added to the *WCETs*. Big-application A has 3 tasks. Task T_1 is an independent task with period $P_1 = 5$ and $e_1 = 2$. Task T_2 is a synchronization task with period $P_2 = 20$ and non-critical section code segment, $e_{2,1} = 2$, critical section code segment $cs_{2,2} = 1$ and non-critical section code segment $e_{2,3} = 8$. Task T_3 is

a synchronization task with period $P_3 = 21$ and non-critical section code segments $e_{3,1} = 4$, critical section code segment $cs_{3,2} = 1$ and non-critical section code segment $e_{3,3} = 14$. Initially, we put all 3 tasks in synchronization core.

However, the total utilization of these 3 tasks is $2/5+11/20+19/21 = 1.85 > 1$. So we move task T_1 to execution core. But task T_3 is still not schedulable since the utilization of the remaining two tasks is $11/20 + 19/21 = 1.45 > 1$. So we move task T_2 to execution core and task T_2 becomes a multicore task.

We transform each multicore task T_i in *exe_core* to $*T_i$:

- Task T_1: ($e_1 = 2; P_1 = 5$) is a single-core task and hence there is no transformation is needed.
- Task T_2: ($e_{2,1} = 2$, $cs_{2,2} = 1$, $e_{2,3} = 8$; $P_2 = 20$) is transformed to $*T_2$:($e_{2,1} = 2$, $R(cs_{2,2}) = 2$, $e_{2,3} = 8$; $P_2 = 20$), where $R(cs_{2,2})$ consists of 1 unit of blocking time and 1 unit of its critical section execution time.

Task T_1 in execution core is the highest priority task with utilization 0.4 and hence it is schedulable. We now check if multicore task $*T_2$ is schedulable.

- $r^0 = R(cs_{2,2}) + e_1 + (e_{2,1} + e_{2,3}) = 2 + 2 + 10 = 14;$
- $r^1 = 2 + 10 + \left\lceil \frac{14}{5} \right\rceil * 2 = 18$
- $r^2 = 2 + 10 + \left\lceil \frac{18}{5} \right\rceil * 2 = 20$
- $r^3 = 2 + 10 + \left\lceil \frac{20}{5} \right\rceil * 2 = 20$

That is, task $*T_2$ finishes at $t = 20$, thus $*T_2$ is schedulable. It follows that T_2 is schedulable. Note that T_2 actually finishes at $t = 19$.

In the synchronization core, task T_3 is a single-core task that can be preempted by task T_2's critical section code.

- $r^0 = cs_{2,2} + (e_{3,1} + cs_{3,2,} + e_{3,1}) = 1 + (4+1+14) = 20;$
- $r^1 = 19 + \left\lceil \frac{20}{5} \right\rceil * 1 = 20$

Thus, task T_3 is also schedulable.

3.0 Summary

This paper extends the SCE approach by creating a new technology called Virtual Single-core (VSC). VSC technology enables users to use a group of cores as if it were a larger single-core chip.

There are many certified avionics software developed for single-core chips. VSC facilitates the migration of single-core avionics software to multicore computers.

References

[1] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transaction on Computers, 1979.

[2] L. Sha, M. Caccamo, R. Mancuso, J-E. Kim, M-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale and R. Bradford, Single-core Equivalent Technology for Hard Real-Time Computing on Multicore Processors, IEEE Computer, September 2016. This is an overview of https://www.ideals.illinois.edu/handle/2142/55672

[3] L. Sha, R. Rajkumar and J. P. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, IEEE Transactions on Computers, September 1990.

[4] J. Lehoczky, L., Sha, L., D.Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Behavior," Proceedings of the IEEE Real-Time System Symposium, 1989, pp. 166 – 171.

[5] L. Sha, L. R. Rajkumar, and S. Sathaye, "Generalized Rate Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," Proceedings of the IEEE, January 1994.

[6] J. Strosnider, J. P. Lehoczky and L. Sha, The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, IEEE Transaction on Computers, January, 1995.

[7] A. Block, H. Leontyev ; B. B. Brandenburg ; J. H. Anderson, A Flexible Real-Time Locking Protocol for Multiprocessors, RTCSA 2007.

[8] B. B. Brandenburg and J. H. Anderson, A Comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT, Principles of Distributed Systems, Volume 5401 of the series Lecture Notes in Computer Science pp 105-124

[9] R. Rajkumar, Real-time synchronization protocols for shared memory multiprocessors. Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 116–123 (1990)

[10] J. H. Anderson, J. M. Calandrino, Real-Time Scheduling on Multicore Platforms, RTAS, 2006.

[11] H. Kim, D. de Niz, B. Andersson, M. Klein, O.

Mutlu, R. Rajkumar: Bounding and reducing memory interference in COTS-based multicore systems. Real-Time Systems 52(3): 356-395 (2016)

[12] H. Yun, R. Mancuso, Z. P. Wu, R. Pellizzoni. "PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platform ", RTAS, 2014

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. "MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms", RTAS, 2013

[14] J.-E. Kim, M.-K. Yoon, R. Bradford, and L. Sha, "Integrated Modular Avionics (IMA) Partition Scheduling with Conflict-Free I/O for Multicore Avionics Systems", in IEEE Computer Software and Applications Conference, July 2014.

[15] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, Heechul Yun, "WCET(m) Estimation in Multi-Core Systems using Single-core Equivalence", ECRTS http://control.lth.se/ecrts2015/ .

[16] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, R. Pellizzoni. “Real-Time Cache Management Framework for Multi-core Architectures”, RTAS, April 2013

[17] FAA Position Paper on Multi-Core Processors, CAST-32 (Rev 0), Online resource available at: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf. Oct. 27, 2014.

[18] C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM, 1973.