

Received September 8, 2019, accepted September 23, 2019, date of publication September 30, 2019, date of current version October 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2944662

Portable and Configurable Implementation of ARINC-653 Temporal Partitioning for Small Civilian UAVs

HYUN-CHUL JO¹, JOO-KWANG PARK¹, HYUN-WOOK JIN¹², (Member, IEEE),
HYUNG-SIK YOON³, AND SANG HUN LEE³

¹Department of Computer and Information Communication Engineering, Konkuk University, Seoul 05029, South Korea

²Department of Computer Science and Engineering, Konkuk University, Seoul 05029, South Korea

³Agency for Defense Development, Daejeon 34186, South Korea

Corresponding author: Hyun-Wook Jin (jinh@konkuk.ac.kr)

This work was supported in part by the South Korea Agency for Defense Development under Grant UD170054JD.

ABSTRACT The ARINC-653 standard defines temporal partitioning that enables multiple avionics applications to execute independently from each other without interference in terms of CPU resources. Though partitioning has been mainly discussed from the viewpoint of manned aircraft, it can also efficiently integrate multiple applications on civilian Unmanned Aerial Vehicles (UAVs) that have even severer limitations on size, weight, power, and cost. In order to employ ARINC-653 temporal partitioning to civilian UAVs, its implementation must be flexible enough to be applied to diverse run-time software environments and computing hardware platforms. In this paper, we suggest a portable and configurable implementation of ARINC-653 for small-sized civilian UAVs aiming for low cost, easy development, and easy extension. Our implementation provides the Operating System (OS) abstraction layer that defines the essential OS-level features and the OS-independent interfaces to the upper layer that actually implements the ARINC-653 standard. Our implementation is also modularized so that the policies of resource management in CPU scheduling and memory allocation can be easily extended and selectively configured. In addition, we implement the advanced resource management schemes to promote the benefits of multi-core processors that are already widely deployed in Commercial Off-The-Shelf (COTS) systems. We show that our ARINC-653 implementation is portable across different OS, such as Linux and RTEMS, reusing the most of source codes thanks to the layered and modular design. We also analyze the overheads of the ARINC-653 APEX interfaces and multi-core scheduling. Moreover, we conduct a case study for a small-sized quad-copter.


INDEX TERMS ARINC-653, integrated modular avionics, multi-core, portability, temporal partitioning, unmanned aerial vehicles.

I. INTRODUCTION

As the number of embedded computing devices in the current-generation avionics systems is growing rapidly, the issues of Size, Weight, Power, and Cost (SWaP-C) become more difficult to resolve. To address SWaP-C, the Integrated Modular Avionics (IMA) architecture that provides an efficient way of integrating several real-time applications on a single computing device has been suggested [1]. Though the IMA architecture is mainly discussed from the viewpoint of large aircraft or manned aerial vehicles, small Unmanned Aerial Vehicles (UAVs) are those that indeed require IMA to reduce SWaP-C. U.S. Army, for example,

categorizes UAVs with a maximum gross takeoff weight of less than 20 *lb* and an airspeed below 100 *kn* as Group 1 (i.e., small UAVs) [2]. As low-cost and small-sized UAVs become popular in various civilian domains, such as hobby, filmmaking and surveillance, it is highly required that various, multiple applications can be easily integrated on a single embedded computer according to their purposes in small-quantity batch production. Moreover, the integrated software tasks have to satisfy the real-time requirements. Thus, it is expected that IMA can provide efficient and reliable consolidation of real-time applications of small-sized civilian UAVs, improving its productivity.

The ARINC-653 standard defines the essential software features for IMA and specifies the APplication/EXecutive (APEX) interfaces (i.e., APIs) [3]. Partitioning is the

The associate editor coordinating the review of this manuscript and approving it for publication was Fuhui Zhou .

key feature defined by ARINC-653 to guarantee several real-time applications to exclusively utilize CPU and memory resources reserved, while providing isolation of execution environments between different applications. In order to employ ARINC-653 partitioning to small-sized civilian UAVs, its implementation must be flexible enough to be applied to diverse run-time software environments and computing hardware platforms. Though there were significant studies on the design and implementation of the ARINC-653 standard [4]–[11], the flexibility was not a focus of the previous researches. Since the majority of small-sized civilian UAVs use Commercial Off-The-Shelf (COTS) software and hardware, which are more varied than those for military or typical aircraft, the ARINC-653 implementation has to be portable across different software environments and configurable according to different platforms. However, different Operating Systems (OS), for instance, implement different task scheduling policies and system calls; thus, an ARINC-653 implementation is very likely to be OS-dependent. Moreover, as multi-core processors are already prevalent in COTS hardware, the ARINC-653 implementation for UAVs must efficiently utilize multiple CPU cores.

In this paper, we suggest a portable and configurable ARINC-653 implementation for small-sized civilian UAVs aiming for low cost, easy development, and easy extension. Our implementation provides the OS abstraction layer that defines the essential OS-level features and OS-independent interface to the upper layer that actually implements the ARINC-653 standard. Thus, we can easily replant the ARINC-653 implementation to another OS by modifying the OS abstraction layer only, while reusing the most of codes that are directly related to ARINC-653. In addition, we implement the advanced resource management schemes to promote the benefits of multi-core processors. The measurement results show that our ARINC-653 design is portable across different OS thanks to the OS abstraction layer and can make the scheduling overheads overlap with application execution on a multi-core processor by the advanced resource management. We also present a case for a small UAV, where real-time applications run on our ARINC-653 implementation. We summarize our contributions as follows:

- We propose an internal design of ARINC-653 aiming for portability and configurability over various run-time environments of civilian UAVs.
- We present the implementation of the new features defined in the ARINC-653 standard to support multi-core processors.
- This study can provide practical insights into requirements of applying ARINC-653 to small civilian UAVs.

The rest of this paper is organized as follows: In Section II, we give an overview of ARINC-653. We suggest an implementation of portable and configurable ARINC-653 in Section III. In this section, we describe the OS abstraction layer and the support for multi-core processors. The performance evaluation of the ARINC-653 implementation is

conducted in Section IV. We discuss related work and the limitations of our study in Sections V and VI, respectively. Finally, we conclude this paper in Section VII.

II. ARINC-653 OVERVIEW

The ARINC-653 standard defines the APEX interfaces between the OS of an avionics computer and the application software [3]. The interfaces allow the application software to control the CPU scheduling and communication.

The ARINC-653 standard defines temporal partitioning that enables the avionics applications to execute independently from each other in terms of CPU resources. This partitioning concept is a key for IMA architecture as it provides the resource isolation between applications (i.e., partitions). The partitions are created at the system initialization phase and cannot be removed or added dynamically. The CPU scheduling algorithm of partitions is predetermined, repetitive with a fixed periodicity, which is represented as partition windows in the configuration file. A partition window is a time duration, for which the CPU resources are given to the specified partition.

A partition comprises one or more processes that share the resources of the partition but are not visible outside of the partition. Each process has a priority level and can be preempted by a higher-priority process. The process scheduling can be either periodic or aperiodic based on the policy configured when each process is created. In general, the partition scheduler and process scheduler are implemented in a hierarchical manner; once the partition scheduler decides which partition to give the CPU resources, the process scheduler then decides which processes of this partition to execute within the given partition window. In this paper, we focus on how to provide the portability and configurability in the implementation of this hierarchical CPU scheduling. In the newest version of the ARINC-653 Part 1 [3], a partition can run across multiple CPU cores, while a process can run on only one of the cores assigned to the partition. The core affinity of a process can be specified by the APEX interface, `INITIALIZE_PROCESS_CORE_AFFINITY`.

Regarding communication, ARINC-653 defines APEX interfaces for intra- and inter-partition communication. The intra-partition communication interfaces that include buffers, blackboards, semaphores, and events provide communication between two or more processes running in the same partition. The buffers store messages in the message queue and deliver to the receiver in FIFO order. On the other hand, the blackboards do not queue messages. Any message written to a blackboard remains there until the message is either cleared or overwritten by a new message. The semaphores are used to synchronize between processes. An event is a communication mechanism that notifies an occurrence of a condition to processes which are waiting for it. The inter-partition communication interfaces provide communication between two or more partitions that may run on different computing devices. Since we target small UAVs that are equipped with a single embedded computer and run independent applications,

in the latter sections, we do not consider the inter-partition communication.

III. ARINC-653 TEMPORAL PARTITIONING

As described in Section II, temporal partitioning provides a framework for safe consolidation of real-time applications with respect to CPU resources. In this section, we suggest a design of ARINC-653 temporal partitioning that is portable and configurable for diverse software and hardware platforms of small civilian UAVs.

A. OVERALL DESIGN

ARINC-653 can be implemented in different software space (e.g., user-, kernel-, and hypervisor-space) according to the requirements of target systems [12]. As small civilian UAVs have a wide choice of COTS software and hardware platforms, providing the portability of the ARINC-653 implementation is very important, but it is not trivial to realize. Aiming for portability, we implement ARINC-653 at user-space and introduce an OS abstraction layer that provides the OS-independent primitive functions essential to the ARINC-653 implementation. Thus, by modifying only the OS abstraction layer, we can apply our ARINC-653 implementation to a different platform. This allows us to reuse the most of source codes already verified. In addition, our ARINC-653 implementation provides the modularity in that the policies of resource management can be newly defined and plugged in as a module for different runtime environments. There were also several user-space implementations of ARINC-653 [4]–[6]; however, the portability of these originated from using the Portable Operating System Interface (POSIX) and was limited to POSIX-compliant OS. Unlike existing implementations, our OS abstraction layer can provide even distinguished portability and we empirically show that our ARINC-653 can run successfully on different OS, such as Linux and RTEMS. We will describe more details of our layered and modular design in Subsection III-B.

The multi-core processors are already deployed in many civilian embedded systems including UAVs. The latest revision of the ARINC-653 standard also includes the extensions to support multi-core processors as described in Section II. However, the actual implementation issues have not been extensively discussed in literature in terms of ARINC-653. Existing researches on ARINC-653 focus more on the application-level issues of multi-core systems, such as analysis of Worst-Case Execution Time (WCET) [13] and decision of core affinity [14]. In this paper, we present an actual implementation of ARINC-653 temporal partitioning for multi-core processors. Our implementation can provide a reference design of how to manage the partition windows and core affinity on multi-core processors. Some of such details are omitted in the ARINC-653 standard. Our ARINC-653 implementation also can be configured to decide where the multi-core schedulers run. Such flexibility enables the scheduling overheads to be overlapped with actual execution

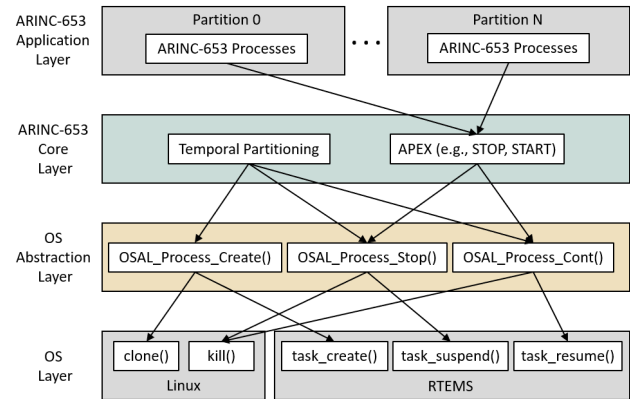


FIGURE 1. Layered design of ARINC-653.

of applications. We will describe more details of multi-core scheduling in Subsection III-C.

B. LAYERED AND MODULAR DESIGN

Our ARINC-653 implementation has a two-layer architecture at the user-space as shown in Figure 1, where the upper layer is the ARINC-653 core layer and the other is the OS abstraction layer. The ARINC-653 core layer implements temporal partitioning and APEX, which are independent on the underlying software and hardware platforms. The OS abstraction layer provides the OS-independent abstraction to the ARINC-653 core layer, so that the implementation of the upper layer can be reused without modifications for different platforms. This can make the modeling and verification of software much easier and improve the safety [15], [16].

For example, the ARINC-653 processes can be scheduled by signaling or direct scheduling calls, depending on the underlying OS. In either case, the ARINC-653 processes are implemented as threads because those belong to the same partition share the address space. In Linux, though signaling is the more practicable mechanism for scheduling of ARINC-653 processes, it is difficult to signal a specific thread belong to a different address space unless each thread has a unique process ID. Thus, we create an ARINC-653 process by calling the `clone()` system call with the `CLONE_VM` flag turned on and the `CLONE_THREAD` flag turned off, so that each thread has its own process ID, sharing the address space. On the other hand, Real-Time Operating Systems (RTOS), such as RTEMS [17], provide not only POSIX-compliant signaling system calls but also native direct scheduling calls that suspend and resume the execution of a task. Since the latter interfaces provide less overheads, we use direct calls, such as `rtems_task_suspend()` and `rtems_task_resume()` in RTEMS, instead of signaling. The OS abstraction layer hides such details by providing simple scheduling interfaces named `OSAL_Process_Stop()` and `OSAL_Process_Cont()` to the upper layer. The current implementation of the OS abstraction layer supports Linux and RTEMS, of which the performance will be presented in Subsection IV-A.

In addition, our ARINC-653 also supports storageless UAVs that may not include a storage, such as flash memory, due to a limited payload or a lightweight RTOS that does not support a file system. The ARINC-653 standard provides the examples of the configuration table in XML schema format, which includes partition windows and attributes of partitions and processes. The embedded systems that have a storage system can store the configuration file in the storage and read it at the run-time to initialize the system. However, if the storage system does not exist, we need a different mechanism to represent the system configuration. Our ARINC-653 implementation can configure the system at either the run-time or the compile time. On UAVs with a storage system, we represent the system configuration in the JSON format, store it as a file, and read the file at the run-time. On the other hand, the system configuration is represented in source codes (i.e., a header file) when the target UAV does not have a storage system; that is, the attributes and scheduling behaviors of partitions and processes are decided at the compile time.

There also can be diverse demands on different policies for resource management, such as CPU scheduling and dynamic memory allocation. To cope with such demands, we manage the functions that implement policies via an array of function pointers. The ARINC-653 process scheduler, for instance, can change its algorithm by changing the corresponding function pointer without side effects to other functionality. In regard to the memory allocation, some platforms may not support dynamic memory allocation, or some others may not want dynamic memory allocation because of run-time overheads and failure. To address these requirements, our modular framework allows the APEX interfaces (e.g., buffer and blackboard services) to statically allocate the internal buffers at the compile time based on the configuration.

C. MULTI-CORE SCHEDULING

As we have described in Subsection III-A, our ARINC-653 is implemented at the user-space, emphasizing portability; that is, the partition and process schedulers are also implemented as user-space daemon processes. Consequently, it gives the system integrator with the flexibility to decide where to run these daemon processes on multiple cores. A general approach is to run a scheduling daemon on each core, for which the daemon is responsible. This approach is quite similar with existing operating systems in that each core performs CPU scheduling on the processes assigned to that core. We call this approach *distributed scheduling* as the scheduling daemons run every core. This approach, however, adds scheduling overheads (i.e., timer, partition scheduling, and process scheduling overheads) between partition windows as shown in Figure 2(a). These scheduling overheads can encroach on the CPU resources reserved by partitions or lag behind in starting partition windows. If the timer is reset when the scheduler is invoked (i.e., (1) in Figure 2(a)), less duration of CPU resources is assigned to partitions than reserved. On the other hand, if the timer is reset at the end

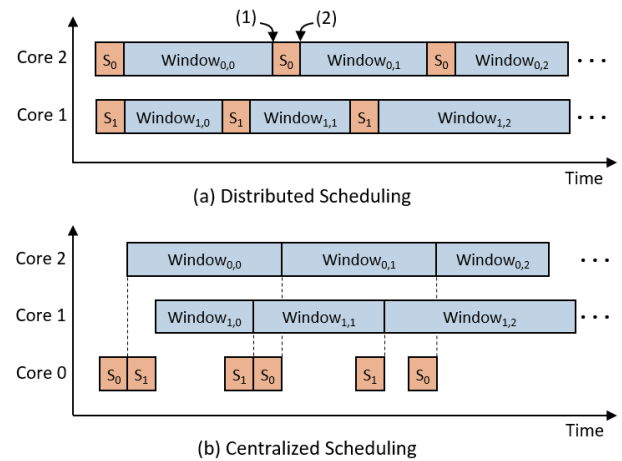


FIGURE 2. Comparison of distributed scheduling with centralized scheduling (s_j represents scheduling overheads).

of the scheduling procedure (i.e., (2) in Figure 2(a)), the time lags are accumulated.

Another approach is to dedicate a core to the scheduling daemons, while application partitions run on the rest of the cores. We call this approach *centralized scheduling*, because a single core takes care of scheduling of the other cores. Although this approach sacrifices a core, as shown in Figure 2(b), the scheduling overheads can be overlapped with partition execution. Consequently, we can eliminate the scheduling overheads that exist between partition windows and remove the side effect of distributed scheduling. In addition, centralized scheduling has the potential to realize the theoretical models for hard real-time scheduling in which the system overheads, such as scheduling overhead, are generally assumed to be zero. Such overlapping of scheduling overheads, however, would be beneficial only for a limited number of cores as the scheduling overheads increase in proportional to the number of cores. Thus, our ARINC-653 provides the system integrator with an interface to select between distributed scheduling and centralized scheduling. We will show the impact of the number of cores on the benefits of centralized scheduling in Subsection IV-B.

We also support the features described in the ARINC-653 standard for multi-core systems. We specify a set of physical cores (Pcores), where a partition can be scheduled, in the configuration file as shown in Figure 3. The Lines 231-241 in this figure represent that Partition 1 can run on Pcores 2 and 3, which are mapped to logical cores (Lcores) 0 and 1 of the partition. Mapping between Pcores and Lcores is performed at the initialization phase by parsing this configuration file. If the system does not support storage system, this mapping is done at the compile time as described in Subsection III-B. In addition, to specify the affinity of processes to Lcores, we add the `processor_core_affinity` entry in the configuration file as shown in Line 12, which represents that Task 1 in Partition 1 has the affinity to the first Pcore (i.e., Pcore 2). Once all processes are created, our initialization daemon initializes the core affinity of each process


```

1  "scheduler_affinity_type": "centralized",
2  "partition": [
3  {
4    "identifier": 1,
5    "name": "Partition1",
6    "entry_point": "create_partition",
7    "process": [
8    {
9      "name": "Task1",
10     "entry_point": "Task1",
11     "scheduler_type": "RM",
12     "processor_core_affinity": 0,
13     "priority": 12,
14     "period": 40,
15     "time_capacity": 10,
16     "deadline": 40,
17     "stack_size": 30720
18     },
19     :
20     :
21     :
22     :
23     :
24     :
25     :
26     :
27     :
28     :
29     :
30     :
31     :
32     :
33     :
34     :
35     :
36     :
37     :
38     :
39     :
40     :
41     :
42     :
43     :
44     :
45     :
46     :
47     :
48     :
49     :
50     :
51     :
52     :
53     :
54     :
55     :
56     :
57     :
58     :
59     :
60     :
61     :
62     :
63     :
64     :
65     :
66     :
67     :
68     :
69     :
70     :
71     :
72     :
73     :
74     :
75     :
76     :
77     :
78     :
79     :
80     :
81     :
82     :
83     :
84     :
85     :
86     :
87     :
88     :
89     :
90     :
91     :
92     :
93     :
94     :
95     :
96     :
97     :
98     :
99     :
100    :
101    :
102    :
103    :
104    :
105    :
106    :
107    :
108    :
109    :
110    :
111    :
112    :
113    :
114    :
115    :
116    :
117    :
118    :
119    :
120    :
121    :
122    :
123    :
124    :
125    :
126    :
127   "schedules": [
128   {
129     "scheduler_core": 2,
130     "partition_time_window": [
131     {
132       "partition_name_ref": "Partition1",
133       "offset": 0,
134       "duration": 10,
135       "periodic_processing_start": "true"
136     },
137     :
138     :
139     :
140     :
141     :
142     :
143     :
144     :
145     :
146     :
147     :
148     :
149     :
150     :
151     :
152     :
153     :
154     :
155     :
156     :
157     :
158     :
159     :
160     :
161     :
162     :
163     :
164     :
165     :
166     :
167     :
168     :
169     :
170     :
171     :
172     :
173     :
174     :
175     :
176     :
177     :
178     :
179     :
180     :
181     :
182     :
183     :
184     :
185     :
186     :
187     :
188     :
189     :
190     :
191     :
192     :
193     :
194     :
195     :
196     :
197     :
198     :
199     :
200     :
201     :
202     :
203     :
204     :
205     :
206     :
207     :
208     :
209     :
210     :
211     :
212     :
213     :
214     :
215     :
216     :
217     :
218     :
219     :
220     :
221     :
222     :
223     :
224     :
225     :
226     :
227     :
228     :
229     :
230     :
231     :
232     :
233     :
234     :
235     :
236     :
237     :
238     :
239     :
240     :
241     :
242     :
243     :
244     :
245     :
246     :
247     :
248     :
249     :
250     :
251     :
252     :
253     :
254     :
255     :
256     :
257     :
258     :
259     :
260     :
261     :
262     :
263     :
264     :
265     :
266     :
267     :
268     :
269     :
270     :
271     :
272     :
273     :
274     :
275     :
276     :
277     :
278     :
279     :
280     :
281     :
282     :
283     :
284     :
285     :
286     :
287     :
288     :
289     :
290     :
291     :
292     :
293     :
294     :
295     :
296     :
297     :
298     :
299     :
300     :
301     :
302     :
303     :
304     :
305     :
306     :
307     :
308     :
309     :
310     :
311     :
312     :
313     :
314     :
315     :
316     :
317     :
318     :
319     :
320     :
321     :
322     :
323     :
324     :
325     :
326     :
327     :
328     :
329     :
330     :
331     :
332     :
333     :
334     :
335     :
336     :
337     :
338     :
339     :
340     :
341     :
342     :
343     :
344     :
345     :
346     :
347     :
348     :
349     :
350     :
351     :
352     :
353     :
354     :
355     :
356     :
357     :
358     :
359     :
360     :
361     :
362     :
363     :
364     :
365     :
366     :
367     :
368     :
369     :
370     :
371     :
372     :
373     :
374     :
375     :
376     :
377     :
378     :
379     :
380     :
381     :
382     :
383     :
384     :
385     :
386     :
387     :
388     :
389     :
390     :
391     :
392     :
393     :
394     :
395     :
396     :
397     :
398     :
399     :
400     :
401     :
402     :
403     :
404     :
405     :
406     :
407     :
408     :
409     :
410     :
411     :
412     :
413     :
414     :
415     :
416     :
417     :
418     :
419     :
420     :
421     :
422     :
423     :
424     :
425     :
426     :
427     :
428     :
429     :
430     :
431     :
432     :
433     :
434     :
435     :
436     :
437     :
438     :
439     :
440     :
441     :
442     :
443     :
444     :
445     :
446     :
447     :
448     :
449     :
450     :
451     :
452     :
453     :
454     :
455     :
456     :
457     :
458     :
459     :
460     :
461     :
462     :
463     :
464     :
465     :
466     :
467     :
468     :
469     :
470     :
471     :
472     :
473     :
474     :
475     :
476     :
477     :
478     :
479     :
480     :
481     :
482     :
483     :
484     :
485     :
486     :
487     :
488     :
489     :
490     :
491     :
492     :
493     :
494     :
495     :
496     :
497     :
498     :
499     :
500     :
501     :
502     :
503     :
504     :
505     :
506     :
507     :
508     :
509     :
510     :
511     :
512     :
513     :
514     :
515     :
516     :
517     :
518     :
519     :
520     :
521     :
522     :
523     :
524     :
525     :
526     :
527     :
528     :
529     :
530     :
531     :
532     :
533     :
534     :
535     :
536     :
537     :
538     :
539     :
540     :
541     :
542     :
543     :
544     :
545     :
546     :
547     :
548     :
549     :
550     :
551     :
552     :
553     :
554     :
555     :
556     :
557     :
558     :
559     :
560     :
561     :
562     :
563     :
564     :
565     :
566     :
567     :
568     :
569     :
570     :
571     :
572     :
573     :
574     :
575     :
576     :
577     :
578     :
579     :
580     :
581     :
582     :
583     :
584     :
585     :
586     :
587     :
588     :
589     :
590     :
591     :
592     :
593     :
594     :
595     :
596     :
597     :
598     :
599     :
600     :
601     :
602     :
603     :
604     :
605     :
606     :
607     :
608     :
609     :
610     :
611     :
612     :
613     :
614     :
615     :
616     :
617     :
618     :
619     :
620     :
621     :
622     :
623     :
624     :
625     :
626     :
627     :
628     :
629     :
630     :
631     :
632     :
633     :
634     :
635     :
636     :
637     :
638     :
639     :
640     :
641     :
642     :
643     :
644     :
645     :
646     :
647     :
648     :
649     :
650     :
651     :
652     :
653     :
654     :
655     :
656     :
657     :
658     :
659     :
660     :
661     :
662     :
663     :
664     :
665     :
666     :
667     :
668     :
669     :
670     :
671     :
672     :
673     :
674     :
675     :
676     :
677     :
678     :
679     :
680     :
681     :
682     :
683     :
684     :
685     :
686     :
687     :
688     :
689     :
690     :
691     :
692     :
693     :
694     :
695     :
696     :
697     :
698     :
699     :
700     :
701     :
702     :
703     :
704     :
705     :
706     :
707     :
708     :
709     :
710     :
711     :
712     :
713     :
714     :
715     :
716     :
717     :
718     :
719     :
720     :
721     :
722     :
723     :
724     :
725     :
726     :
727     :
728     :
729     :
730     :
731     :
732     :
733     :
734     :
735     :
736     :
737     :
738     :
739     :
740     :
741     :
742     :
743     :
744     :
745     :
746     :
747     :
748     :
749     :
750     :
751     :
752     :
753     :
754     :
755     :
756     :
757     :
758     :
759     :
760     :
761     :
762     :
763     :
764     :
765     :
766     :
767     :
768     :
769     :
770     :
771     :
772     :
773     :
774     :
775     :
776     :
777     :
778     :
779     :
780     :
781     :
782     :
783     :
784     :
785     :
786     :
787     :
788     :
789     :
790     :
791     :
792     :
793     :
794     :
795     :
796     :
797     :
798     :
799     :
800     :
801     :
802     :
803     :
804     :
805     :
806     :
807     :
808     :
809     :
810     :
811     :
812     :
813     :
814     :
815     :
816     :
817     :
818     :
819     :
820     :
821     :
822     :
823     :
824     :
825     :
826     :
827     :
828     :
829     :
830     :
831     :
832     :
833     :
834     :
835     :
836     :
837     :
838     :
839     :
840     :
841     :
842     :
843     :
844     :
845     :
846     :
847     :
848     :
849     :
850     :
851     :
852     :
853     :
854     :
855     :
856     :
857     :
858     :
859     :
860     :
861     :
862     :
863     :
864     :
865     :
866     :
867     :
868     :
869     :
870     :
871     :
872     :
873     :
874     :
875     :
876     :
877     :
878     :
879     :
880     :
881     :
882     :
883     :
884     :
885     :
886     :
887     :
888     :
889     :
890     :
891     :
892     :
893     :
894     :
895     :
896     :
897     :
898     :
899     :
900     :
901     :
902     :
903     :
904     :
905     :
906     :
907     :
908     :
909     :
910     :
911     :
912     :
913     :
914     :
915     :
916     :
917     :
918     :
919     :
920     :
921     :
922     :
923     :
924     :
925     :
926     :
927     :
928     :
929     :
930     :
931     :
932     :
933     :
934     :
935     :
936     :
937     :
938     :
939     :
940     :
941     :
942     :
943     :
944     :
945     :
946     :
947     :
948     :
949     :
950     :
951     :
952     :
953     :
954     :
955     :
956     :
957     :
958     :
959     :
960     :
961     :
962     :
963     :
964     :
965     :
966     :
967     :
968     :
969     :
970     :
971     :
972     :
973     :
974     :
975     :
976     :
977     :
978     :
979     :
980     :
981     :
982     :
983     :
984     :
985     :
986     :
987     :
988     :
989     :
990     :
991     :
992     :
993     :
994     :
995     :
996     :
997     :
998     :
999     :
1000    :

```

FIGURE 3. Example of the configuration file in JSON format.

according to the affinity and mapping information by calling the `INITIALIZE_PROCESS_CORE_AFFINITY` APEX. We assume that the core affinity of processes is not dynamically changed at the run-time. We also introduce other attributes for multi-core, which are not defined in the ARINC-653 standard. The `scheduler_affinity_type` entry in Line 1 represents the affinity policy of scheduling daemons and can have either centralized or distributed as described above. The `scheduler_type` entry in Line 11 specifies the process scheduling policy as described in Subsection III-B. To define partition windows for each core, the `scheduler_core` entry is inserted as shown in Line 129. The following lines (i.e., from Lines 130 to 136) represents the first partition window on Pcore 2.

IV. PERFORMANCE EVALUATION

In this section, we show the potential of our ARINC-653 implementation along with detailed performance evaluation and a case study for small UAV.

A. OVERHEADS OF THE APEX INTERFACES

In this subsection, we analyze the overheads of APEX interfaces on Linux and RTEMS, respectively. Our intention is,

TABLE 1. APEX overheads on Linux.

APEX interfaces	Ave. (μ s)	Dev.
GET_PARTITION_STATUS	33.8	10.8
GET_PROCESS_STATUS	29.7	10.6
GET_MY_ID	2.4	0.5
GET_PROCESS_ID	0.9	0.5
GET_MY_PROCESSOR_CORE_ID	5.0	1.0
INITIALIZE_PROCESS_CORE_AFFINITY	14.0	3.1
START	22.4	2.3
STOP	52.5	3.9
CREATE_PROCESS	63.8	33.7

TABLE 2. APEX overheads on RTEMS.

APEX interfaces	Ave. (μ s)	Dev.
GET_PARTITION_STATUS	15.3	1.4
GET_PROCESS_STATUS	15.2	1.4
GET_MY_ID	1.2	0.4
GET_PROCESS_ID	2.0	0.6
GET_MY_PROCESSOR_CORE_ID	1.2	0.4
INITIALIZE_PROCESS_CORE_AFFINITY	1.1	0.3
START	14.3	1.6
STOP	19.8	9.7
CREATE_PROCESS	81.0	17.3

however, not to compare the performance on different OS but to show that the OS abstraction layer can efficiently support different OS for the ARINC-653 core layer. The lines of code (LOC) of the common source codes for two different OS is about 6,500, while the OS-dependent codes in the OS abstraction layer are about 1,100 LOC for Linux and 1,700 LOC for RTEMS. That is, we could reuse 79 ~ 85% of source codes.

We conducted the overhead measurement on an Intel i5 system running Ubuntu 16.04 (Linux kernel version 4.15) and an ARM Cortex-A7 system running RTEMS (kernel version 4.11.3). Tables 1 and 2 show the measurement results with the APEX interfaces for partition and process management. We can observe that the first two interfaces (i.e., `GET_PARTITION_STATUS` and `GET_PROCESS_STATUS`) show high overheads on both Linux and RTEMS as these interfaces internally exchange messages with the scheduling daemons to get the partition status and process status. We see higher overheads on Linux due to the trap exception for messaging system calls, but the RTEMS case does not show an overhead as much as Linux because RTOS generally provide the system services through direct function calls. The next three interfaces in each table (i.e., `GET_MY_ID`, `GET_PROCESS_ID`, and `GET_MY_PROCESSOR_CORE_ID`) show very low overheads since the process IDs and the core IDs are stored in the global memory of the corresponding partition. That is, these interfaces simply refer to that memory area to get the ID information. The `INITIALIZE_PROCESS_CORE_AFFINITY` interface initializes the core affinity of the specified process by calling the system calls, such as `sched_setaffinity()` on Linux and `rtems_task_set_affinity()` on RTEMS, of which the overheads are shown in the tables. The `START` and `STOP` interfaces resume and suspend the execution of

a process, respectively. Thus, these interfaces need to exchange messages with scheduling daemons to change the state of the process, which results in high overheads. The STOP interface on Linux especially has to send a SIGSTOP signal to the corresponding process if it is in the running state. The CREATE_PROCESS interface shows the highest overhead in the tables since it calls a complex system call, such as `clone()` and `rtms_task_create()`, to create a new process in the system.

B. SCHEDULING OVERHEAD

In this subsection, we compare the scheduling overhead of centralized scheduling with that of distributed scheduling. To analyze the scheduling overhead, we measured the time interval between the start of the current partition window and that of the next partition. The closer this time interval is to the duration of the partition window in the configuration file, the fewer side effects due to the scheduling overhead are. Thus, we reported the difference between the time duration in the configuration file and the time interval measured for each partition window. A negative means that the actual system assigns less CPU resources to the partition than configured; on the other hand, a positive number represents that CPU resources are assigned to the partition more than configured or leaked by the scheduler. We conducted the experiments on an Intel i7 hexa-core system, where CentOS 7 (Linux kernel version 3.10) was installed. We analyzed the scheduling overhead on dual-, quad-, and hexa-core cases as these are popular numbers of cores in COTS multi-core processors. We ran two partitions for each core, and their period and duration were configured as 25 ms and 10 ms, respectively. Each partition comprised two CPU-intensive tasks, the period and the time capacity of each task were set to 50 ms and 10 ms, respectively. We measured 100 samples of time intervals for consecutive partition windows.

The measurement results are shown in Figures 4, 5, and 6, which depict the differences between reserved and actual time duration of partition windows. Thus, the closer to zero, the more accurate the resource was provided. In the case of dual-core, we can observe that centralized scheduling shows very little jitters (i.e., $-7.6 \mu\text{s} \sim +6.5 \mu\text{s}$), while distributed scheduling shows larger deviation and jitters (i.e., $-89.8 \mu\text{s} \sim +71.5 \mu\text{s}$). The jitters of centralized scheduling become worse as the number of cores increases, but those of distributed scheduling is not affected significantly. Consequently, distributed scheduling shows less jitters than those of centralized scheduling in the hex-core system (Figure 6). That is, centralized scheduling can effectively make the scheduling overheads overlap with the execution of partitions; however, its overheads increase in proportional to the number of cores and worsen the jitters.

C. CASE FOR DRONE

To validate the operability of our ARINC-653 implementation, we performed a case study for small drone. We used the Erle-Copter [18], of which the dimension was

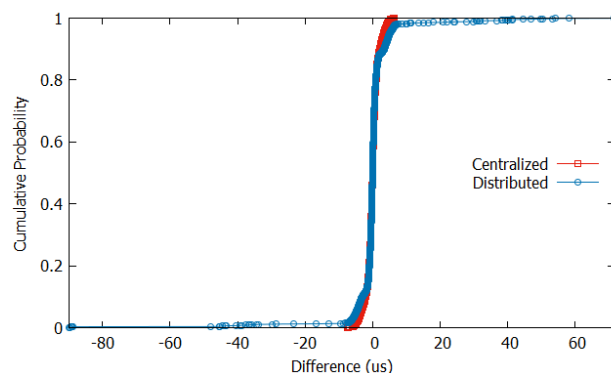


FIGURE 4. Differences between reserved and actual partition windows on a dual-core processor.

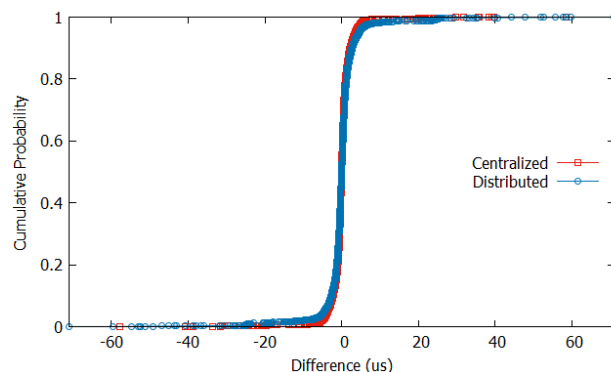


FIGURE 5. Differences between reserved and actual partition windows on a quad-core processor.

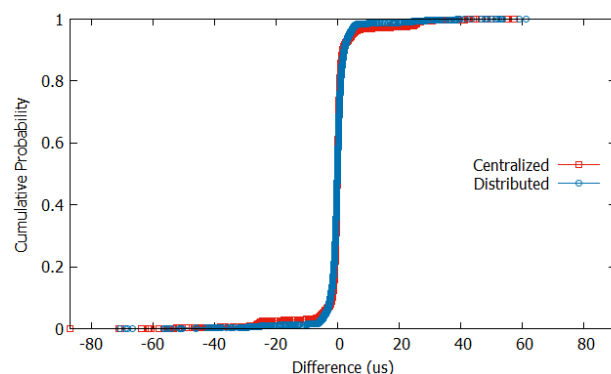


FIGURE 6. Differences between reserved and actual partition windows on a hexa-core processor.

370 x 370 x 95 mm and the payload was 1 kg. This quadcopter was equipped with an Erle-Brain 3 flight control computer that consisted of a Raspberry Pi 3 Model B embedded board, IMU, servo controller, etc. We installed Raspbian (Linux kernel version 4.4.50) and our ARINC-653 implementation on the Raspberry Pi 3. We ran two application partitions that performed flight control and video streaming, respectively. ArduPilot [19] was modified to implement the flight control partition, which consisted of six processes: i) The Timer process read the sensing data from the IMU (Inertial Measurement Unit) sensor and updated the attitude information in the blackboard; ii) The RCIN process received

TABLE 3. Period and time capacity of processes.

Partition	Process	Period (ms)	Time Capacity (ms)
Flight Control	Timer	1.0	0.3
	RCIN	0.5	0.2
	Main	2.5	2.4
	UART	10.0	0.2
	I/O	20.0	0.3
	ToneAlarm	10.0	0.2
Video Streaming	VStreaming	33.0	13.2

TABLE 4. Period and duration of partitions.

Partition	Physical Core	Period (ms)	Duration (ms)
Flight Control	1	2.5	2.4
	2	1.0	0.7
	3	5.0	0.7
Video Streaming	3	33.0	23.1

control signal from radio control transmitter and stored it to the blackboard; iii) The Main process read the sensing and control data from the blackboard, performed noise filtering, and controlled motors; iv) The UART process read GPS information; v) The I/O process logged sensor data to microSD; and vi) The ToneAlarm process turned on or off LEDs based on the current state. In the modified ArduPilot, these processes followed the ARINC-653 periodic process model and used ARINC-653 intra-partition communication interfaces, such as blackboards and semaphores. The video streaming partition had only one periodic process that sent video frames of 320 x 240 in MJPEG format via wireless LAN. The frame rate was 30 fps. We decided the period and time capacity of the processes as shown in Table 3.

The core affinity assigned to each process is shown in Figure 7. Since the Raspberry Pi 3 board in our experimental system had a quad-core processor, we used the centralized scheduling policy that showed less jitters than distributed scheduling on a quad-core processor as discussed in Subsection IV-B. Thus, the Pcore 0 was dedicated to the scheduling daemons. The flight control partition ran across the other three cores (i.e., Pcores 1 to 3). The video streaming partition ran only on Pcore 3, which was shared with the flight control partition. The period and duration of the partitions were calculated as shown in Table 4 by using the algorithm suggested by Shin and Lee [20].

We conducted an indoor flight test with respect to take off, hovering, rotating, and landing. The left picture of Figure 8 shows the snapshot of the flight test. The right picture is the video frame received at that point.

To show the benefits of using temporal partitioning in this scenario, we measured the frames per second provided by the VStreaming process, while running it together with another partition that burned 20% of CPU resources. In this experiment, we added this artificial partition, because the three tasks of the flight control partition that ran with VStreaming in Figure 7 hardly consumed the CPU resources. As we can see in Figure 9, we can guarantee the 30 fps with ARINC-653, but the number of frames drops significantly without ARINC-653 because of interference with the background partition.

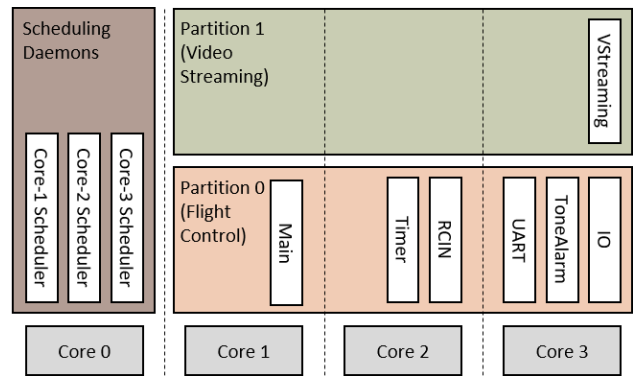


FIGURE 7. Core affinity of application partitions and scheduling daemons.

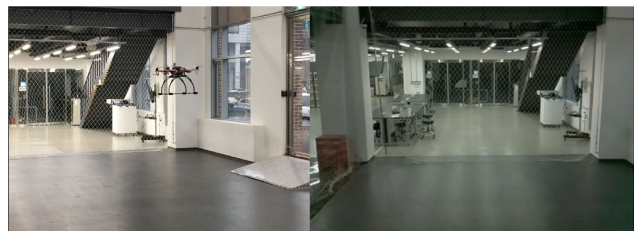


FIGURE 8. Indoor flight test.

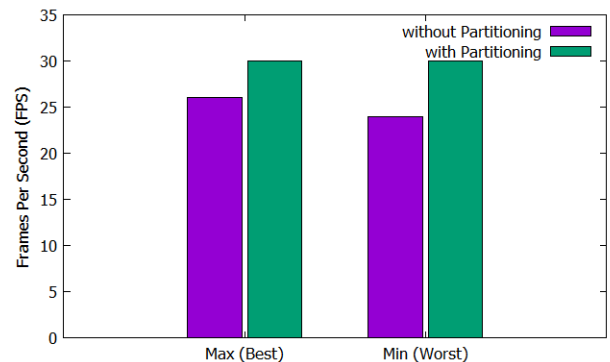


FIGURE 9. Benefits of temporal partitioning.

This clearly shows that the temporal partitioning of ARINC-653 can isolate the execution environment between partitions and provide transparent consolidation of different avionics applications (i.e., partitions).

V. RELATED WORK

Many industrial and academic organizations implemented the ARINC-653 standard. Most of commercial products and few academic instances [11], [21] were implemented at the kernel-space; thus, these showed very low overheads but had less flexibility and portability [12]. A challenging approach was to employ the virtualization technology for partitioning [7]–[9]. This virtualization-based implementation allowed different OS to run simultaneously on a single embedded computing board; however, the implementations at the hypervisor-space was not portable and added significant system overheads. There were also user-space designs aiming for portability [22]. For example, AMOBA [4] and SIMA [5] were developed for the purpose of simulating

the ARINC-653 run-time environment and implemented on top of POSIX. In addition, the CORBA Component Model with ARINC 653 [6] defined a component-based model for ARINC 653 and was also implemented on POSIX. However, since not all OS are fully POSIX-compliant, the portability of these implementations is very limited. Our ARINC-653 implementation includes the OS abstraction layer that provides better portability than existing implementations. Consequently, we could show that our ARINC-653 successfully ran on Linux and RTEMS with minimal modifications.

The Open Group Future Airborne Capability Environment (FACE) provides an open standard platform for avionics software applications. The Operating System Segment (OSS) of the FACE reference architecture defines the interfaces that should be provided by the FACE-compliant OS to provide application portability. These interfaces are a combination of the ARINC-653 and POSIX standards. That is, the interfaces defined by the OSS are higher-level interfaces than the OS abstraction layer in our implementation. It is also noteworthy that our ARINC-653 implementation can promote Linux and RTEMS to FACE-compliant OS because Linux and RTEMS support POSIX.

There were also significant discussions on the ARINC-653 temporal partitioning over multi-core processors. Huyck [23] classified the ARINC-653 partition scheduling for multi-core processors into Asymmetric Multi-Processing (AMP) and Symmetric Multi-Processing (SMP). In AMP, all processes in a partition run on the same core, while, in SMP, processes of a partition can run on different cores. Jean *et al.* [24] and Pathan *et al.* [25] especially focused on AMP systems in their studies. Carrascosa *et al.* [9] implemented both AMP and SMP on XtratuM. Silva and Tatibana [10] also implemented AMP and SMP on AIR. However, discussions on how to optimize ARINC-653 itself on multi-core systems in terms of scheduling overheads were not focused. Our ARINC-653 implementation not only supports both AMP and SMP, but also prepares different policies named distributed and centralized scheduling to reduce the scheduling jitters according to the number of cores.

Overall, our portable and configurable design and implementation of ARINC-653 is more suitable for small civilian UAVs equipped with diverse COTS platforms compared with existing researches.

VI. DISCUSSION

In this section, we discuss the limitations of our study in terms of compliance to the ARINC-653 specification. The ARINC-653 specification is organized into four parts, Part 1 [3] of which defines the required services such as partition management, process management, time management, memory management, inter-partition communication, and intra-partition communication. In this paper, we have mainly referred to Part 1 and suggested an implementation that supports the full features of temporal partitioning and intra-partition communication, while omitting memory partitioning and inter-partition communication. Besides temporal

partitioning, memory partitioning is another indispensable feature to guarantee isolated run-time environments between partitions. The memory management service in Part 1 limits the memory usage of each partition and stack size of each task, which are supposed to be specified in configuration file. Our implementation partially supports memory partitioning with the aid of OS. In both Linux and RTEMS, we can specify the stack size for each task. It, however, is difficult to limit the total memory usage of each partition in our implementation. Regarding spatial security, the memory access across different partitions is prohibited by means of memory protection provided by the Linux kernel and MMU of processors. On the other hand, RTEMS uses a flat memory model and all processes in RTEMS are basically implemented as threads that share the same address space; thus, it likely happens that a task that is either erroneous or malicious accesses the memory area of another partition.

The inter-partition communication service defined in Part 1 provides communication between two or more partitions that may run on the same node or different nodes. The inter-partition communication can operate in either queuing mode or sampling mode. In the queuing mode, the messages are queued in the internal message queue and passed to the application in a FIFO manner. Thus, there is no intentional message loss in this mode. On the other hand, in the sampling mode, only the last message is saved overwriting the previous one. Since we target partitions that are not correlate, inter-partition communication is not supported in our current implementation. If we consider communication between partitions that only run on the same embedded computer, we believe that our implementation can be easily extended to support the queuing and sampling modes of inter-partition communication by using message queue and shared memory IPCs provided OS.

The last part of the ARINC-653 specification, Part 4 [26] defines a subset of the APEX interfaces of Part 1 for a simpler execution model. For the sake of simplicity, Part 4 targets the partitions with only one or two processes (one for periodic and the other for aperiodic) and does not support intra-partition communication and multi-core processors. Such restricted interfaces may be suitable for small UAVs due to smaller footprint and lower overheads. With regard to temporal partitioning, Part 4 does not require a complex real-time scheduler, because it allows only one periodic process within a partition and only one partition window per period. The video streaming partition in Subsection IV-C, for example, has only one periodic process and does not require intra-partition communication; thus, services defined in Part 4 are sufficient. Nevertheless, some of existing UAV control programs consist of several processes as described in Subsection IV-C; thus, one periodic process and one aperiodic process may not be enough to implement a high-end flight control program. If we allow several periodic processes within a partition accordingly, it is required to support multiple partition windows and intra-partition communication. Moreover, since the COTS hardware is equipped with

multi-core processors as we have emphasized in this paper, we believe that supporting multi-core processors will be an important requirement for civilian UAVs. In this, we believe it is valuable to provide both Part 1 and Part 4 for developers and integrators to choose according to the requirements of applications and the capability of hardware platform.

Part 1 also supports health monitoring by providing configuration tables and APEX interfaces for error handling. Errors are classified into process-, partition-, and module-level errors. The recovery actions for partition- and module-level errors are specified in the configuration table. The process-level errors are handled by an error handler implemented by the application developer. In Part 4, only partition- and module-level errors are considered. Our current implementation does not support health monitoring; however, as future work, this should be considered for resilient and secure flight control that avoids crashes and injuries in urban areas or crowded areas.

VII. CONCLUSION

Since civilian UAVs in small-quantity batch production have diverse requirements on software and hardware platforms, it is very critical to provide a portable and flexible run-time environment to applications. In this paper, we suggested a layered and modular design of ARINC-653 for civilian small-sized UAVs. Our ARINC-653 implementation is comprised of the ARINC-653 core layer and the OS abstraction layer for portability. In addition, our implementation provides the modularity so that the system integrator can easily change the policy in resource managements, such as CPU scheduling and memory allocation. Moreover, our ARINC-653 implementation supports multi-core processors and especially provides the distributed and centralized scheduling policies for less jitters on different number of cores.

We showed that our ARINC-653 implementation was portable across different OS (i.e., Linux and RTEMS), reusing the most of source codes (79 ~ 85%) due to the portable and configurable design. We also analyzed the overheads of the APEX interfaces and multi-core scheduling. In addition, we conducted indoor flight tests of a small-sized quad-copter, where we ran two partitions across multiple cores with centralized scheduling.

REFERENCES

- [1] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Proc. IEEE/AIAA DASC*, Dallas, TX, USA, Oct. 2007, pp. 2.A.1-1-2.A.1-10.
- [2] *U.S. Army Unmanned Aircraft Systems Roadmap 2010-2035: Eyes of the Army*, U.S. Army UAS Center of Excellence, Fort Rucker, AL, USA, 2010.
- [3] *Avionics Application Software Standard Interface (Part 1): Required Services*, document ARINC Specification 653P1-4, 2015.
- [4] E. Pascoal, J. Rufino, T. Schoofs, and J. Windsor, "AMOBAs-ARINC 653 simulator for modular based space applications," in *Proc. DASIA*, Palma, Spain, 2008, Art. no. 43.
- [5] T. Schoofs, S. Santos, C. Tatibana, and J. Anjos, "An integrated modular avionics development environment," in *Proc. IEEE/AIAA DASC*, Orlando, FL, USA, Oct. 2009, pp. 1.A.2-1-1.A.2-9.
- [6] A. Dubey, G. Karsai, and N. Mahadevan, "A component model for hard real-time systems: CCM with ARINC-653," *Softw., Pract. Exper.*, vol. 41, no. 12, pp. 1517-1550, Nov. 2011.
- [7] S. H. VanderLeest, "ARINC 653 hypervisor," in *Proc. IEEE/AIAA DASC*, Salt Lake City, UT, USA, Oct. 2010, pp. 5.E.2-1-5.E.2-20.
- [8] S. Han and H.-W. Jin, "Full virtualization based ARINC 653 partitioning," in *Proc. IEEE/AIAA DASC*, Seattle, WA, USA, Oct. 2011, pp. 7.E.1-1-7.E.1-11.
- [9] E. Carrascosa, M. Masmano, P. Balbastre, A. Crespo, and J. Galizzi, "Multicore software architectures on virtualized partitioned systems," in *Proc. DASIA*, Warsaw, Poland, 2014, Art. no. 27.
- [10] C. Silva and C. Tatibana, "Multima—Multi-core in integrated modular avionics," in *Proc. DASIA-Data Syst. Aerosp.*, Warsaw, Poland, 2014, Art. no. 39.
- [11] S. Han and H.-W. Jin, "Kernel-level ARINC 653 partitioning for Linux," in *Proc. ACM SAC*, Trento, Italy, 2012, pp. 781-786.
- [12] S. Han and H.-W. Jin, "Resource partitioning for integrated modular avionics: Comparative study of implementation alternatives," *Softw., Pract. Exper.*, vol. 44, no. 12, pp. 1441-1466, Dec. 2014.
- [13] S. H. VanderLeest, J. Millwood, and C. Guikema, "A framework for analyzing shared resource interference in a multicore system," in *Proc. IEEE/AIAA DASC*, London, U.K., Sep. 2018, pp. 1-10.
- [14] P. Bretault, N. Chatonnay, and B. Calmet, "Approaching parallelization of payload software application on ARM multicore platforms," in *Proc. DASIA*, Barcelona, Spain, 2015, Art. no. 25.
- [15] H. Wang, D. Zhong, T. Zhao, and F. Ren, "Integrating model checking with SysML in complex system safety analysis," *IEEE Access*, vol. 7, pp. 16561-16571, 2019.
- [16] K. Zhang, J. Wu, C. Liu, S. S. Ali, and J. Ren, "Behavior modeling on ARINC653 to support the temporal verification of conformed application design," *IEEE Access*, vol. 7, pp. 23852-23863, 2019.
- [17] The RTEMS Project. *RTEMS Real Time Operating System*. Accessed: May 7, 2019. [Online]. Available: <http://www.rtems.org/>
- [18] Erle Robotics. *Erle-Copter*. Accessed: May 7, 2019. [Online]. Available: http://docs.erlerobotics.com/erle_robots/erle_copter
- [19] ArduPilot Dev Team. *ArduPilot—Open Source Autopilot*. Accessed: May 7, 2019. [Online]. Available: <http://ardupilot.org/>
- [20] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 30:1-30:39, Apr. 2008.
- [21] C. Kown, D. Kim, H. Joe, and H. Kim, "Linux-based memory efficient ARINC 653 partition scheduler," in *Proc. IEEE ETFA*, Barcelona, Spain, Sep. 2014, pp. 1-5.
- [22] Q. Kang, C. Yuan, X. Wei, Y. Gao, and L. Wang, "A user-level approach for ARINC 653 temporal partitioning in seL4," in *Proc. ISSSR*, Shanghai, China, Oct. 2016, pp. 106-110.
- [23] P. Huyck, "ARINC 653 and multi-core microprocessors—Considerations and potential impacts," in *Proc. IEEE/AIAA DASC*, Williamsburg, VA, USA, Oct. 2012, pp. 6.B.4-1-6.B.4-7.
- [24] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, "Ensuring robust partitioning in multicore platforms for IMA systems," in *Proc. IEEE/AIAA DASC*, Williamsburg, VA, USA, Oct. 2012, pp. 7A4-1-7A4-9.
- [25] R. M. Pathan, F. Hashi, P. Stenström, L.-G. Green, T. Hult, and P. Sandin, "Temporal partitioning on multicore platform," in *Proc. DASIA*, Warsaw, Poland, 2014, Art. no. 32.
- [26] *Avionics Application Software Standard Interface (Part 4): Subset Services*, document ARINC Specification 653P4, 2012.



HYUN-CHUL JO received the B.S. degree in computer science and engineering and the M.S. degree in computer and information communication engineering from Konkuk University, Seoul, South Korea, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree in computer and information communication engineering.

His research interests include real-time systems, operating systems, and embedded systems.



JOO-KWANG PARK received the B.S. degree in computer science and engineering from Konkuk University, Seoul, South Korea, in 2017, where he is currently pursuing the M.S. degree in computer and information communication engineering.

His research interests include operating systems, embedded systems, and cloud computing.



HYUNG-SIK YOON has been a Principal Researcher with the Agency for Defense Development (ADD), Daejeon, South Korea, since 1992. His research interest includes safety critical software development and testing.



HYUN-WOOK JIN (M'04) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University, Seoul, South Korea, in 1997, 1999, and 2003, respectively.

From 2003 to 2006, he was a Research Associate with the Department of Computer Science and Engineering, The Ohio State University, USA. Since 2006, he has been with the Department of Computer Science and Engineering, Konkuk University, Seoul, where he is currently a Full Professor.

His research interests include operating systems, real-time computing, and parallel computing.



SANG HUN LEE received the B.S. degree in computer science from Virginia Military Institute, VA, USA, in 2013, and the M.S. degree in computer software from Korea University, Seoul, South Korea, in 2016.

He is currently a Researcher with the Agency for Defense Development (ADD), Daejeon, South Korea. His research interest includes avionic software development and testing.

...