IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 26, 2021, accepted March 14, 2021, date of publication April 7, 2021, date of current version April 16, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3071500

The Migration of Engine ECU Software From Single-Core to Multi-Core

JUN YOUNG MOON[1], DO YEON KIM[1], JIN HO KIM[2], AND JAE WOOK JEON[1], (Senior Member, IEEE)
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 440-746, South Korea
[2]Department of Computer Engineering, Kyungnam University, Changwon 51767, South Korea

Corresponding author: Jae Wook Jeon (jwjeon@yurim.skku.ac.kr)

This work was supported by the Industrial Strategic Technology Development Program (Development of Front/Side Camera Sensor for Autonomous Vehicle) funded by the Ministry of Trade, Industry, & Energy (MI, South Korea) under Grant 10080011.

ABSTRACT As multiple functions have been added to single-core-based engine electronic control units (ECUs) in vehicles, automotive researchers and manufacturers have actively studied multi-core architecture for engine ECUs. Multi-core architecture can provide load balancing and parallelism that can meet the requirements of international organization standard (ISO) 26262. However, since real-world engine ECUs have the most complex automotive open system architecture (AUTOSAR)-based control logic and datasets among automotive ECUs, developing multi-core-based engine ECUs is a substantial amount of work. Thus, automotive researchers and manufacturers will need new methodologies for multi-core-based engine ECUs. In this paper, we focus on designing a multi-core migration methodology and applying it to a real-world AUTOSAR-based engine ECU from HYUNDAI. We verify its practicability and enhanced performance. In conclusion, through connection with other automotive domain ECUs, it is demonstrated that a multi-core engine ECU using our migration technology can be applied in real-world automotive vehicles, leading to a significant improvement in performance.

INDEX TERMS Multi-core-based engine system, shared data inconsistency, core load balancing, memory and offset optimization technology.

I. INTRODUCTION

Recently, the amount of data that must be processed by an engine electronic control unit (ECU) and the number of automotive functions embedded in an engine ECU have increased due to addition of new ECUs and advancements in vehicles. These lead to an increased core load in the engine ECU and an increased need for additional computing power [1]. It is particularly difficult to add new software because the core loads of existing single-core-based engine ECUs have reached their limit [2]. To solve these problems, automotive manufacturers and researchers have considered multi-core processors. Multi-core architecture can reduce the computation power and core load. Additionally, such architecture can provide parallelism, which is needed to satisfy international organization standard (ISO) 26262 [1]. However, 100% redesigned multi-core-based engine ECUs should not be considered because of large monetary and time costs. Thus, a methodology of migrating from existing single-core to multi-core architecture is needed. Reliable migration technology can reduce costs and increase time efficiency compared with redesigning the multi-core system. In addition, this approach can help integrate the accumulated engine ECU software knowledge and architecture.

Migration to an actual multi-core-based engine is not a simple process; an engine ECU has many datasets, complex automotive open system architecture (AUTOSAR)-based control logic, and hard-real-time characteristics [3]–[5] (These figures are the highest in automotive ECUs). Further, there are noteworthy issues derived from the above studies.

First, a safe and efficient shared data protection method considering an engine ECU should be researched. Shared data occur when a piece of data stored in one memory is accessed by different cores, as shown in Fig. 1. If shared data are used without protection, the stability and coherency of the shared data will not be protected, and the shared data can be altered dramatically when read or written. To address these issues, a previous study presented shared data protection using

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

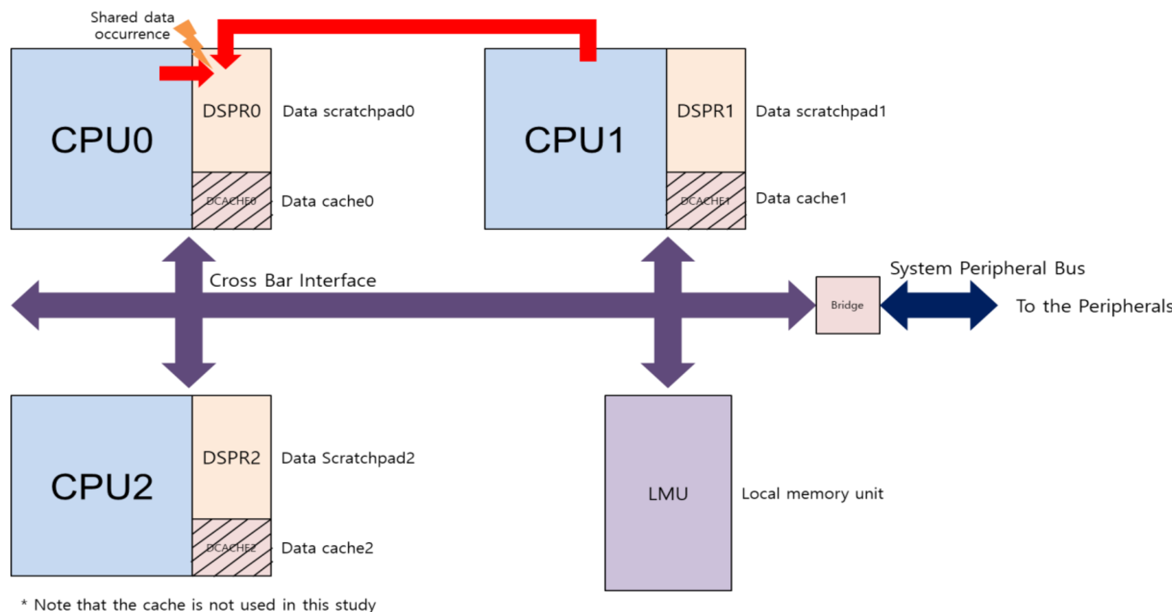

FIGURE 1. The example of MCU structure using in multi-Core Engine ECU.

an explicit synchronization and wait-free and semaphore locks method as an automotive multi-core platform [6]. The research in [7] studied several shared data protection methods and presented an algorithm for choosing an appropriate method. However, this method is difficult to apply in an actual engine ECU. The approach in [6], [7] is to apply a protection method for each set of shared data. Thus, if the number of shared data items increases, the length of the protection code becomes proportionally larger. Therefore, existing methods are not suitable for an engine ECU, which processes a large amount of shared data. As another shared data problem, an automated tool is necessary to apply the consistency method to an actual ECU. Due to the large amount of shared data in an engine ECU, it is impossible to manually secure the consistency of the shared data. Therefore, it is necessary to analyze and implement the functions of a tool that can automatically insert the shared data inconsistency method into the existing engine ECU control logic [8]. However, this is a critical issue for developers because there is currently no tool that can automatically analyze the control logic and insert shared data inconsistency measures.

Next, elaborated core load balancing technology for an engine ECU should be considered. Core load balancing comprises allocation of tasks to each core for shared use [9]. To apply such technology, the data to the engine ECU should be allocated statically because AUTOSAR [10], [11] does not allow dynamic allocation [3]. In [12]–[16], the authors researched the static task allocation method to balance the core load. However, these previous studies [12]–[16] did not consider the additional core load of the shared data protection method. In addition, there are various types of tasks in an engine ECU [5], and it is necessary to consider the different loads of various tasks. However, in [12]–[16], they did not consider all types of tasks. Thus, accurate core load balancing results are unlikely.

Finally, optimization technologies of multi-core tasks that can improve task scheduling performance should be considered. The first optimization scheme is to allocate data to memory. Since the data access time of each core changes according to the memory allocation position of the data in the multi-core [13], which has multiple memory areas [17], the execution time of a task can be changed according to the data position within the memory. Because the task execution time change affects the load of the core, if a systematic memory allocation algorithm is constructed, the core load and task execution time can be reduced. However, existing research does not present a specific static memory allocation method considering the shared and non-shared data of engine ECUs. The second optimization scheme concerns the offset of each task, which determines the start position of a task. An efficient offset allocation method can reduce the response time and collisions between tasks [18]. However, despite the potential advantages, no offset allocation method considering an AUTOSAR-based engine ECU operation has been studied.

To solve the aforementioned problems, we propose key methodologies for shared data protection, core load balancing, and optimization. The migration methodology from single core to multi-core is performed at the source code level of the application. We apply these to an actual engine ECU from HYUNDAI and verify the practicability and enhanced performance using hardware in the loop simulation (HILs), which can provide information and messages from other domain ECUs in real-world automotive vehicles [19].

This paper is organized as follows. Section II introduces the details of the developed migration technology method. Section III implements and analyzes the proposed migration technology, and Section IV concludes this paper.

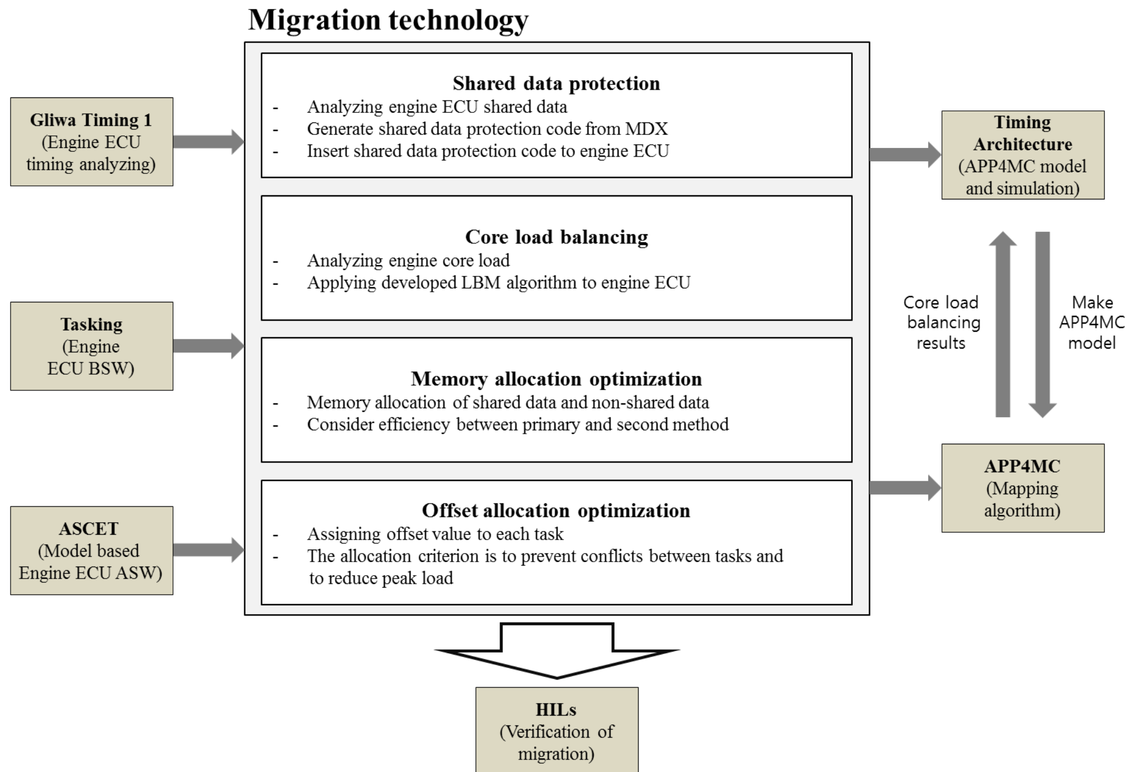

FIGURE 2. The concept of the migration technology method using shared data protection, core load balancing, optimization of memory and offset allocation and multiple tools (Timing 1, Tasking, ASCET, TA, APP4MC and HILs) for the multi-core based engine ECU.

II. MULTI-CORE MIGRATION TECHNOLOGY

In this section, we suggest a shared data protection scheme, load balancing, and a method of optimizing migration to an actual multi-core-based engine ECU. Fig. 2 shows the concept of our migration technology. We used multiple tools to apply the migration method. Tasking and ASCET are tools that can modify the engine ECU control logic [20], [21]. Tasking manages AUTOSAR-based basic software (BSW), and ASCET is used for modeling the engine ECU application software (ASW) [5], which is based on the Motor Industry Software Reliability Association (MISRA)-C: 2004 [22]. The details of Timing 1, Timing Architecture (TA), HILs, and the Application Platform Project for Multi-core (APP4MC) platform are introduced in Section IV. Our migration technology is compatible with the above tools, and the detailed functions of our technology are as follows.

A. SHARED DATA PROTECTION

The AUTOSAR-based ECU uses data through implicit communication of runnables. Before runnable start, a data buffer used in the runnable is copied to a local runnable buffer, which is written back to the existing data buffer after runnable end [23]. Maintaining data consistency has been demonstrated through the runnable's copy strategy of implicit communication [24]. In other words, since AUTOSAR-based runnables only use data from the local runnable buffer, their design has been established so that they are not affected by other runnable operations. Vehicle developers consider the characteristics of the above implicit communication to design vehicle software. Therefore, data consistency was maintained in the single-core-based ECU using single-core implicit communication. However, when migrating to multi-core, a number of shared variables occur. Existing AUTOSAR does not describe a data consistency methodology of shared variables.

When each core of the multi-core accesses shared data without the protection method, the data can cause race conditions. Two major problems about race conditions are important to multi-core technologies for stability and coherency [8]. Therefore, a protection method is necessary to ensure the absence of race conditions. However, because there is so much shared data in an engine ECU, the core execution time (CET) significantly increases when an inefficient protection method is used. To solve this issue, we formulate an optimal race condition protective solution.

First, we investigate three types of shared data in an engine ECU: shared data read-read, shared data write-write, and shared data read-write. Next, we consider use of the lock method in the multi-core-based engine ECU to protect the coherency. The lock-free, wait-free, and lock-key methods have been considered to secure data consistency. The lock-free method can write data using the compare and swap (CAS) command that has the advantage of reducing latency because it uses an atomic command. However, it has a disadvantage in that it can be used only for writing data.

The wait-free scheme has the advantage of ensuring integrity without a lock key because it uses a read/write memory buffer. However, the memory capacity can reach its limit since the engine ECU has many write commands. The lock-key method is based on the spinlock methodology, which can support read/write cases of shared data. Additionally, since spinlock does not use the memory resources, it is an appropriate method for use in engine ECUs. Therefore, we use spinlock in the lock method.

After choosing the spinlock method, we consider how to apply this method efficiently, since overuse of spinlocks increases the task execution time of engine ECUs. To address this, we consider the copied shared data method. First, this method converts the name of existing shared data into a task-specific name. If the existing shared data name is shared_data_1, which is used in the 1ms task, this method changes the name to shared_data_1_T1; this means that this data is only used in the 1ms task. Using the same principle, if the 10ms task also has shared_data_1, the name will be changed to shared_data_1_T10. In other words, global shared data are converted to local shared data unique to each task. By converting local shared data, this method can avoid usage of a global variable, which is recommended in MISRA-C: 2004 [22]. After changing the name, the critical copy/update section is inserted into the normal section. Here, we define the critical section as the control logic protected from the spinlock and the normal section as the control logic not protected from the spinlock. At the start of the critical copy/update section, all the interrupts should be disabled, and the spinlock request function should be inserted. Conversely, the spinlock release function should be inserted at the end of the critical copy/update section, and all interrupts should be re-enabled. The critical copy section replicates the original shared data before the task starts. For example, shared_data_1_T1 copies the shared_data_1 value. After copying, the task executes the normal section using the generated copied shared data. Using the local shared data in the normal section secures the stability of race conditions. When the normal section of the task is completed, the value of the copied shared data is updated to the original shared data via the update critical section before the end of the task. Likewise, shared_data_1 is updated to the shared_data_1_T1 value. Here, if the data are of the read type in a task, they are not updated to the original global value. Since read type data do not use stored instructions, they do not need to be updated to the original global value. Our copy/update shared data method uses just two locks (which are in the copy and update critical sections), resulting in reduced remote blocking time [25] and CET. This approach considers an efficient coherency method by reducing the number of spinlocks and source code level protection methods. As described above, data consistency through the copy strategy of implicit communication of the runnables has been demonstrated, allowing data consistency of the local shared variables in the task. Therefore, if we use our shared variable copy/update strategy, we can ensure safe write of shared data in critical and non-critical sections.

To apply the coherency method, automated tools are needed because the engine ECU uses a significant amount of shared data. However, no tools have been applied in the above coherency method. Thus, we develop automated tools for our coherency method for application in engine control logic and describe the methodology below.

1) THE COPY AND UPDATE CRITICAL SECTION MAKING TOOL

The goal of this tool is to generate prototypes of the critical copy and update sections. The operation of this tool is described in detail as follows. First, the tool must extract shared data information and shared data read/write information. If the shared data are read type, the tool will insert the data into the critical copy section. In contrast, if the shared data are write type, the tool will insert the data into the copy and update the critical section. To create these two files, this tool requires a meta data exchange (MDX) format and uses a core allocation file as the input data.

- The MDX file: MDX is a software architecture format defined through cooperation with automotive companies [26] (Bosch, Audi, Volkswagen, Continental, etc.). It contains ECU software architecture information, such as tasks and runnables, and data information of the ECU like number of data items in a task or runnable. However, the format of existing MDX files is difficult to apply to multi-core environments because existing MDX is based on single-core environments. For example, there is no protocol that distinguishes between global and local shared data including copied shared data. To solve this problem, we modified the MDX file format to be suitable for multi-core systems and distinguish between global and local shared data using unused MDX fields. (<LONG>, <ADMIN-DATA> in MDX) [26].
- Core allocation file: This file has information that specifies the tasks to be distributed by each core. The core allocation information of this file can be used to find data shared between tasks located in different cores. This file is created via core load balancing as described in Section III.

Using the modified MDX file and the core allocation file, the tool extracts the shared data information file of each task and the shared data read/write type file of each task as the output. Additionally, the tool uses the two files to create the copy/update source codes comprising a header file and C files. In the C files, the copy/update functions and spinlock functions of each task are defined. In the header file, declaration of the original shared data and the copied shared data is included, as is that of the critical copy and update section functions and the spinlock functions.

2) ENGINE SOFTWARE CONVERSION TOOL

After creating prototypes of the shared data copy and update functions, these must be applied to the existing engine ECU software. Our engine software conversion tool performed this work through two major functions. First, at the beginning and

end of each task, the copy and update functions are added. In addition, to allow use of these added functions, a new header file for the critical section is added to the engine source code file. Second, the existing shared data in each task are converted into copied shared data. The input of this tool requires the existing ECU source code and copy/update functions source code from the critical copy/update section tool.

B. CORE LOAD BALANCING

Existing tasks of engine ECU have different execution times and periods. When migrating to a multi-core, the load value of each core changes according to allocation. To solve this, first, we need to define multiple types of tasks in the engine ECU. The types of tasks in current engine ECUs are divided into periodic, aperiodic, and sporadic tasks as established by HYUNDAI [5]. Periodic tasks are invoked at the same fixed time intervals, while aperiodic tasks occur at unknown intervals. Sporadic tasks do not show set periodicity, similar to aperiodic tasks, but the period changes according to the revolutions per minute (RPM) of the engine. That is, if the engine crankshaft is running faster, the period of the sporadic task becomes shorter. Second, the core load of the newly added shared data protection code should be considered. Since the number of shared data items is different according to core allocation, the core load of shared data protection code varies depending on core allocation. To produce sophisticated core load balancing techniques, shared data protection code load of each allocation case should be calculated accurately. We could consider a heuristic search for our methodology which could derive faster calculations, although their accuracy would not be guaranteed. However, once the software architecture of an engine ECU is determined, it will be used until the model is discontinued. Therefore, even if our algorithm takes more time than the heuristic method, its usage is more preferable since it achieves greater accuracy. The one-time measurement result of our algorithm is used for years, allowing vehicle consumers to drive their vehicles in a more secure environment. For that reason, we developed load balancing of migration (LBM), the protocol of which is as follows:

1. Under LBM, two properties of the core are analyzed in the first step. 1) The processing speed of each core is measured as each can support heterogeneous speeds. 2) The number of cores in the MCU (m) is calculated to produce cases where tasks are allocated to specific cores.

- m = number of cores in the MCU

2. In the second step, the information of sporadic, periodic, and aperiodic tasks should be analyzed. We extract the average execution time of all tasks and assume that sporadic tasks are periodic. Unlike aperiodic tasks, sporadic tasks have shared variables, and they exhibit the same minor characteristics of periodic tasks. The CET and period of sporadic tasks were assumed to be the average values. The required information is as follows. 1) The normal section execution times of tasks (τ_{normal}) and aperiodic tasks (A_k): in the second step, the algorithm simply analyzes the normal section of tasks but not the critical section of tasks. Furthermore, we suppose that aperiodic tasks do not have a critical section since the engine ECU does not have shared data. If an aperiodic task has shared data in another ECU, Equation 4 can be used to find the critical section. Creating a critical section in the interrupts is nearly impossible because of the irregular occurrence time. 2) The numbers of tasks (n) and aperiodic tasks (l): n and l values are used in the fourth step when making the allocation case.

- n = number of tasks
- τ_{normal} = normal section execution time of task j
- l = number of aperiodic tasks
- A_k = execution time of aperiodic task k

3. The third step analyzes the data name and the read/write directions of all tasks. To calculate the critical section execution time of each allocation case, LBM needs to know the number of shared data items. To calculate this number, it is necessary to compare the data names of tasks in different cores and analyze the data read/write direction. Here, we assume that the read/read relation between data is not shared.

4. The LBM solver distinguishes allocating tasks and aperiodic tasks as follows.

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1, \ldots n \tag{1}$$

$$\sum_{i=1}^{m} y_{ik} = 1, \quad k = 1, \ldots l \; x_{ij}, y_{ik} \in 0, 1 \tag{2}$$

$$Case_P = (x_{01}, x_{02} \ldots x_{(m-1)j}, y_{01}, y_{02} \ldots y_{(m-1)k}) \quad p = 1, \ldots m^{j+k} \tag{3}$$

- x_{ij} = allocation value of task j and core i
- y_{ik} = allocation value of aperiodic task k and core i
- $Case_P$ = a case p of x_{ij} and y_{ik}

The case considered here is the grouping of x_{ij} and y_{ik}, like in Equations (1) and (2), and the LBM solver gathers results for all cases of x_{ij} and y_{ik}. If x_{ij} is 1, task j is allocated in core i. Otherwise if x_{ij} is 0, task j is not allocated in core i. The y_{ik} value is allocated in the same way. If the difference in load ratio between cases exceeds 10%, the case is excluded. The allocation result of a case is the same as in Equation (3).

5. Using the allocation results, we can calculate the task total execution time (τ_j^p). τ_j^p is the sum of the execution times (τ_{normal}) of the critical and normal sections. τ_{normal} was calculated in the second step; thus, we only need to know the task j critical section execution time of each case $p(\tau_{critical}^{pj})$, which is the sum of the shared data copy time ($v_c^{pj} T_{copy}$), update time ($v_u^{pj} T_{update}$) and spinlock time (T_{lock}). This is calculated in Equation (4). Here, the copy/update time of a shared data (T_{copy}, T_{update}) is the copy/update process time of one shared data. T_{lock} is the added value of the lock request instruction processing time and release time instruction processing time. After calculating this, τ_j^p is determined by Equation (5).

$$\tau_{critical}^{pj} = v_c^{pj} T_{copy} + v_u^{pj} T_{update} + T_{lock} \tag{4}$$

$$\tau_{normal} + \tau_{critical}^{pj} = \tau_j^p \tag{5}$$

- $\tau_{critical}^{pj}$ = CET of a task *j* critical section in $Case_P$
- v_c^{pj} = the number of shared data of task *j*, which should be copied in $Case_P$
- v_u^{pj} = the number of shared data of task *j*, which should be updated in $Case_P$
- T_{copy} = copy time of a shared data
- T_{update} = update time of a shared data
- T_{lock} = sum of the lock request and release time
- τ_j^p = the total execution time of task *j* in $Case_P$

The total core execution time of core *i* in $Case_P(C_i^p)$ is calculated in Equation (6). Additionally, the maximum core execution time of each case (C_{max}^p) is determined in Equation (7). C_{max}^p is used in the sixth step, and is compared with execution times of the other cores.

$$\sum_{j=1}^{n} \tau_j^p x_{ij} + \sum_{k=1}^{l} A_k y_{ik} = C_i^p, \quad i = 0, \cdots m \quad (6)$$

$$max(C_0^p, C_1^p \ldots C_i^p) = C_{max}^p \quad (7)$$

- C_i^p = total execution time of core *i* in $Case_P$
- C_{max}^p = the longest total execution time of a core in $Case_P$

6. In the final step, the optimal and most balanced case (C_{result}) is chosen in Equation (8). To accomplish this, the minimum value of C_{max}^p is calculated. The C_{result} case has the shortest core execution time and is considered optimal. The suboptimal value is the second smallest C_{max}^p value.

$$min(C_{max}^1, C_{max}^2 \ldots C_{max}^{m^{j+k}}) = C_{result} \quad (8)$$

- C_{result} = selected optimal and balanced case

The time complexity of the task allocation algorithm is O[2N], and the space complexity is O[N]. The value of N corresponds to the number of task.

C. OPTIMIZATION OF THE MULTI-CORE ECU SOFTWARE

Optimization techniques are required for efficient migration. In this paper, optimized memory allocation and task offset techniques are described. Memory allocation is a technology for multi-core micro controller units (MCUs), which have multiple random access memories (RAM), and includes an allocation strategy description of all data of the engine ECU. Using this memory allocation technology, the existing CET of all tasks can be reduced. Task offset technology assigns an offset to all periodic tasks. Using offset technology, the preemption, response time, and peak load between tasks can be reduced. We coded the optimization technique and created it with a tool.

1) MEMORY ALLOCATION OPTIMIZATION

Each core of multi-core has its own memory access time that is very fast; however, the memory access times of the other cores are relatively slow. Hence, each core shows a different CET performance according to the memory allocation method of the data. The types of data that are involved in the memory allocation technique are shared data, non-shared data, and local shared data. Here, non-shared data and local shared data are not used by other cores, so these data are allocated in the RAM of each core. To address bank conflict, we stack data from the lowest address and accumulate shared data from the first address of the remaining memory. To allocate shared data, the memory access times of each core and the shared data read/write information are needed.

- SDR_N^K = *N* th shared data read count when allocated in core *K*
- SDW_N^K = *N* th shared data write count when allocated in core *K*
- ACR_R^K = *R* th RAM read access time of core *K*
- ACW_R^K = *R* th RAM write access time of core *K*

TABLE 1. Shared data-memory table.

	RAM 1	RAM 2	⋯
shared-data 1	$\sum_{K=0}^{m-1} SDR_1^K * ACR_1^K + \sum_{K=0}^{m-1} SDW_1^K * ACW_1^K$	$\sum_{K=0}^{m-1} SDR_1^K * ACR_2^K + \sum_{K=0}^{m-1} SDW_1^K * ACW_2^K$	
shared-data 2	$\sum_{K=0}^{m-1} SDR_2^K * ACR_1^K + \sum_{K=0}^{m-1} SDW_2^K * ACW_1^K$	$\sum_{K=0}^{m-1} SDR_2^K * ACR_2^K + \sum_{K=0}^{m-1} SDW_2^K * ACW_2^K$	
⋯	⋯	⋯	

Here, the SDR_N^K and SDW_N^K values must consider the periodic properties of the tasks. Also, if the same bank is used between (non-shared data, local shared data) and shared data, (memory access time + bank collision and row buffer latency) value is added to the ACR_R^K and ACR_R^K. Using these values, we perform a mathematical analysis of the shared-data memory table as shown in Table 1. Each equation represents the total access time when the *N* th shared data are allocated in the *R* th RAM. In Table 1, the RAM location of the value in $min\,(\sum_{K=0}^{m-1} SDR_1^K * ACR_1^K + \sum_{K=0}^{m-1} SDW_1^K * ACW_1^K,\ \sum_{K=0}^{m-1} SDR_1^K * ACR_2^K + \sum_{K=0}^{m-1} SDW_1^K * ACW_2^K)$ becomes the optimization value. The sub-optimal value is the second smallest value.

The next step involves selecting the absolute minimum total access time and the secondary minimum total access time of each shared data item. After selection, the difference between these two total access times is calculated based on RAM allocation priority. That is, when the difference value is the largest in a shared data set, the data should be assigned the highest priority in a RAM and has the minimum total access time. The largest difference indicates that the efficiency of allocation to the RAM, which shows the secondary minimum total access time, is much lower. In addition, if some RAM (which shows the minimum total access time to a shared data) is full, then the allocation priority can be newly determined by comparing the difference between the secondary and third minimum values. This can be performed iteratively as needed. The data is transferred between the different memories using the regular load/store instructions of the cores. Our memory allocation tool creates a memory map containing information about the allocation location of shared data. The memory map is applied when reprogramming the ECU. The time complexity of memory allocation algorithm is O[CS], and the space complexity is O[CS]. The value of C corresponds to the

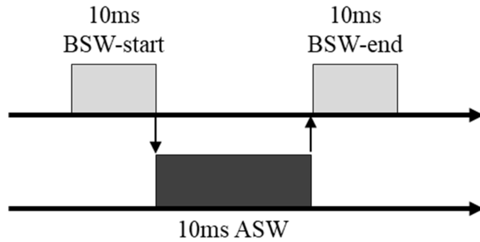

FIGURE 3. AUTOSAR based 10ms task operation of engine ECU.

number of cores, and the value of S corresponds to the number of shared data items.

In addition, we can think of the cache as a memory optimization option. The cache is a high-speed data storage tier for subsets of data. In Fig. 1, data cache exists in each core (DCACHE0, DCACHE1, DCACHE2). Caching is a method of fetching and accessing data stored in main memory into cache. Also, if necessary, the cache should update the data in the main memory or read the latest data from main memory. Cache usage is very easy to implement in the single-core case; however, hardware support technology is required in multi-core-based logic, such as snooping- or directory-based technologies [27]. If the multi-core-based MCU does not provide hardware-based cache coherence technology, cache miss problems and cache coherence problems can be generated. To address this, software-based cache invalidate or flush code must be inserted when using a shared variable. Cache invalidate reads the latest value of main memory by invalidating the cache, and cache flush writes back the value to the main memory as a new value.

2) OFFSET ALLOCATION OPTIMIZATION

Offset allocation is a technique for improving the scheduling performance of tasks by delaying the start point of each task. Using offset technology, the response time, start delay time, and preemption count can be decreased [28]. Response time refers to the execution time from the time the task is activated by the scheduler to the time the task is terminated. Start delay is the wait time between the time the task is activated and the time the task enters the running state. Further, through offset technology, the peak load factor can be reduced. Currently, automotive companies consider the performance of the peak load [1], which is the highest value of the instantaneous core load per 10ms. If the instant core load is high, the patterns of tasks are not uniform, and the operation of tasks is saturated within a certain time. In addition, the engine ECU is related to safety; if the peak load is high, it is likely that urgent tasks will not be executed on time.

Currently, AUTOSAR-based engine ECUs operate in the manner of the BSW-ASW-BSW chain task phase, as shown in Fig. 3, and the period of each periodic task is only determined at BSW-start [5], [29]. BSW-start reads the information needed from ASW, and BSW-end updates the results used by ASW. It is easy to allocate an offset considering only the operation of ASW, but it is very complicated to assign the offset assignment value of each task in the AUTOSAR-based

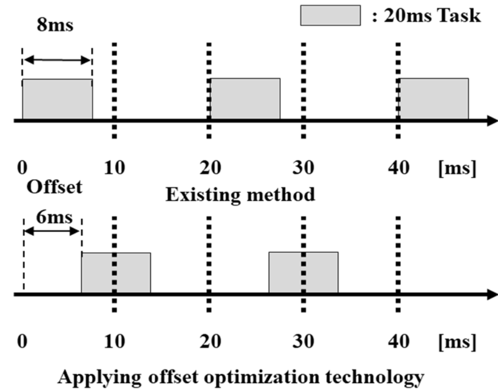

FIGURE 4. Example of 20ms task ASW applying the existing method(above) and the offset allocation method (below).

operation. Considering the above problems, we developed an offset protocol-based AUTOSAR and its protocol-based automation offset tool as follows. First, we divided tasks into two groups: a task group with a period of 10ms or less and a task group with a period exceeding 10ms. This was performed because the offset allocation criteria of the two groups are different. A task exceeding a 10ms period is assigned an offset value to reduce the peak load. Our method is shown in Fig. 4 with an example of a task with a period of 20ms and a CET of 8ms. Before applying the offset technology, the [0, 10] ms moment load is 80%, and the [10, 20] ms moment load is 0%. Therefore, the peak offset before applying the technology is 80%. After applying our offset method, the [0, 10] ms moment load is 40%, and the [10, 20] ms moment load is 40%. After applying the technology, the peak offset is 40%. In this way, our tool deploys tasks exceeding a 10ms period in 10ms unit increments and assigns offset values to avoid collisions. In other words, the midpoint of ASW tasks exceeding 10ms period should be allocated in a 10ms unit. If collisions are inevitable, they should be arranged for minimization. Tasks with 10ms or smaller periods are assigned an offset value to minimize the response time. The allocation method obeys the following rules. A task with a 10ms or smaller period should minimize the number of collisions with other tasks with priorities higher than its own. In the same way, it also should consider conflicts with other tasks with priorities lower than its own. If a task is preempted by the long CET of another task, the offset value minimizing the preemption should be considered as the best case. Our tool considers all cases by incrementally changing the offset of each task with a period less than or equal to 10ms by 10μ s and calculates the number of preemptions of each case.

To select the optimal case, that with the smallest number of preemptions is chosen. The above concept is the same as Equation 9.

$$P_d = \{C_1^{off_x} \dots C_m^{off_x}\} \tag{9}$$

- $m =$ the number of tasks
- $C_m^{off_x} =$ offset value of task m

TABLE 2. Specification of the TC275 MCU.

MCU specification	
Number of cores	3
Processing Speed	Core 0 < Core 1, 2
Main memory	4 (DSPR0, DSPR1, DSPR2, LMU)
Cache usage	Hardware cache coherence

- P_d = Number of preemptions in each offset case
- d = case number

$$\min_{d \in 0 \sim z} (P_d) \tag{10}$$

- z = total number of P_d

Equation (10) is the optimum, and the second smallest is the suboptimal value. The time complexity of the offset allocation algorithm is O[N], and the space complexity is O[N].

III. IMPLEMENTATION AND ANALYSIS

In this paper, we apply our migration technology to the single-core software of a HYUNDAI engine ECU. The single-core software was verified for stability and correct operation and has been used in actual HYUNDAI vehicles. The target multi-core MCU is Tricore TC275, which was developed by the Infineon Company [30]. The specifications of TC275 are described in Table 2. TC275 has three cores with heterogeneous processing speeds [30]. Core 0 is a low-specification core, so its processing speed is slower than those of cores 1 and 2. Further, each core has its own RAM and cache. (DSPR 0, 1, 2, Dcache 0, 1, 2) Additionally, local memory unit (LMU) memory is connected to all cores. This type of memory has lower performance than DSPR, so the access time of LMU RAM is slower. We did not use a cache in our methodology since TC275 does not support hardware-based cache coherence. We could consider using a software cache coherence implementation, but our methodology uses shared variables only in critical sections within the shared variable copy/update. If a cache was used, the latest value of the global shared variable would be copied to local shared variable through invalidate at task startup, and the main memory would have to be accessed directly. Also, at the end of the task, the value of the local shared variable would have to be written back to the main memory as the value of the global shared variable through flush. Even if we used a cache, we would have to access the main memory, negating any advantage of using a cache. Also, our methodology aims to set local shared variables and non-shared variables in their own core RAM memory. According to the TC275 datasheet [30], the core's own RAM read/write access time and the access time through a cache are almost equal to 0 clock cycles. Therefore, use of a cache was not considered in our methodology. The engine ECU software architecture is based on AUTOSAR. Since AUTOSAR is scheduled using a fixed priority mechanism [31], we adopt a rate-monotonic method, which is a pre-emptive static-scheduling technology [32]. In the rate monotonic process, all tasks have fixed priority,

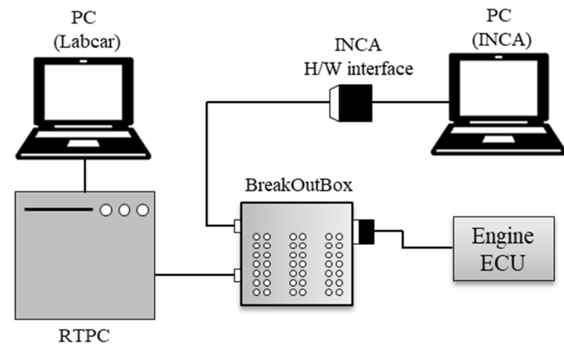

FIGURE 5. HILs environment for verifying engine ECU.

and the priority of a task is higher when its period is shorter. The priority of each task is as follows: Interrupts > 1ms > 2ms > knk > sync0 > sync1 > 5ms > 10ms > 20ms > 50ms > 100ms > 200ms > 1000ms. Each task has the following number of runnables (1ms = 60, 2ms = 48, 5ms = 40, 10ms = 488, 20ms = 62, 50ms = 14, 100ms = 59, 200ms = 88, 1000ms = 30, knk = 44, SyncS0 = 97, SyncS1 = 90). Here, knk, sync0, and sync1 tasks are sporadic with priorities between 2ms and 5ms. The knk task is to assess the engine knocking task in which fuel is abnormally burned in the cylinder of an internal combustion engine. Sync 0, 1 are tasks for synchronizing wheel. The runnables in each periodic task consist of functions that should calculate the data values at each period. In addition, the runnables of sporadic tasks consist of functions that are related to the RPM. We used HILs to verify our engine ECU, which is applied to migration technology. A schematic of the experimental environment is shown in Fig. 5. INCA is a tool of the ETAS Company, and is used to modify the calibration values of the engine ECU, like the offset [33]. Labcar and RTPC are virtual automotive testing and validating systems of the ETAS Company that can change the variables of the automotive operation environment, such as RPM [19]. These tools can transmit message frames of other ECUs to the engine ECU to produce a real automotive operation environment. When HILs were verified, RPMs in the range of [100, 3000] were applied uniformly for 30 min. The total source code size of the software was about 1G bytes, and the binary/object code size of our methodology was 34K bytes. To apply our migration technology, we first analyzed the existing single-core engine ECU. For analysis, we used Timing 1, a timing analysis tool for ECUs developed by the GLIWA Company. This tool analyzes ECU operation in real time as well as the scheduling performance of periodic, sporadic, and aperiodic tasks considering the CET, response time, worst case response time, and spinlock cost [34]. In the next step, we applied our load balancing technique based on the results of the Timing 1 tool. Since the tool that creates the critical copy/update section needs a core allocation file, the shared data protection method is used after applying the core load balancing method. When load balancing technology is applied, the TA tools of VECTOR and APP4MC are utilized. The size of the engine ECU software is so large that it takes a long time to download new software. When verifying the

TABLE 3. Core load balancing results of conventional method using GA and our proposed method using LBM.

Task	Existing avg. core load	# of shared data (copy)	# of shared data (update)	Additional core load	Total avg. core load	GA allocation result	LBM allocation result
1ms	2.68%	40	35	0.41%	3.09%	Core 2	Core 2
2ms	1.32%	85	22	0.30%	1.62%	Core 2	Core 2
5ms	1.18%	79	42	0.13%	1.31%	Core 2	Core 2
10ms	29.7%	3530	2834	3.22%	32.92%	Core 0	Core 0
20ms	4.53%	142	52	0.05%	4.58%	Core 2	Core 2
50ms	0.31%	165	124	0.03%	0.34%	Core 0	Core 2
100ms	2.73%	1152	894	0.10%	2.83%	Core 2	Core 2
200ms	3.3%	3270	2752	0.15%	3.45%	Core 2	Core 2
1000ms	0.78%	1163	1018	0.01%	0.79%	Core 2	Core 2
knk	0.68%	143	96	0.25%	0.93%	Core 2	Core 2
Sync0	7.23%	179	132	0.32%	7.55%	Core 2	Core 2
Sync1	6.12%	182	125	0.32%	6.44%	Core 2	Core 2
Interrupt	0.18%	-	-	-	0.18%	Core 0 (68) Core 2 (93)	Core 0 (83), Core 2 (78)

core load balance, we additionally used TA simulations; this saves a great deal of time if we conduct verification using an accurate TA model before reprogramming the new control logic based on core equalization results. The Timing-1 results are transferred to the TA model in TA. In addition, the created TA model is compatible with APP4MC [35, 36] and is used to execute our LBM algorithm. Additionally, we consider AUTOSAR-based software architecture that consists of the BSW-ASW-BSW chain of tasks [29]. We allocate all BSWs in core 1. BSW should not be distributed to reduce the use of spinlock and code size and is controlled by one core. On the contrary, all ASWs are allocated in cores 0 and 2. Therefore, only ASW is applied to LBM.

When calculating C_i^p, the execution times of the tasks are used as the average value. The number of shared data items and the read/write relationships of each task are obtained through the MDX file. The shared data read/write access time is assumed to be stored in the LMU to balance the access time. The measured core load balancing results are shown in Table 3 for our LBM algorithm and the existing genetic algorithm (GA) load balancing algorithm [35].

The existing GA assigns 10ms and 50ms tasks to core 0 and the remaining tasks to core 2. In our proposed LBM, only a 10ms task is allocated to core 0, while 83 aperiodic tasks are allocated to core 0. The other aperiodic tasks are allocated to core 2. Since sporadic and aperiodic tasks are not considered in the existing scheme, we suppose that the aperiodic tasks are allocated to equalize the results after sporadic and periodic allocation. When simulating the existing GA results in the TA, with 33.7% in core 0 and 32.0% in core 2, a 1.7% load equalization difference was observed. When the existing GA results are applied to the actual engine ECU, the difference in load equalization is 2.1%, 35.2% in core 0 and 33.1% in core 2. When the LBM results were simulated in the TA, core 0 occupied 33.2% and core 2 32.9%, showing a 0.7% load equalization difference. When the LBM result is applied to the actual engine ECU, a 0.3% load equalization difference was observed, with 34.3% in core 0 and 34.0% in core 2. Our developed LBM showed a better equalization effect for the TA model and the actual engine ECU results. Since the verification method using TA is confirmed to have a similar tendency to the actual engine results, subsequent experimental results are shown only as HILs results.

Using the LBM core allocation results and shared data information, we implement a shared data protection strategy to protect against the race condition. The number of copy/update data items is the same as the number of shared data items presented in Table 3. The results of shared data protection are analyzed after applying optimization technology. To use an engine ECU with shared data protection technology more efficiently, we applied the memory optimization technique of shared and non-shared data. In our memory allocation optimization method, data are mapped first to the target platform DSPR. If the priority is constantly being updated and the free capacity of the DSPR is insufficient, the data are set to be allocated to the LMU with the worst performance. The number of extracted non-shared data items was 25326, and the number of shared-data items from MDX was 3530. The existing method is difficult to manage because of the numbers of shared and non-shared data items, which are allocated randomly without a predetermined protocol. When applying our memory optimization allocation technique, non-shared data are locally shared and are allocated to the DSPR of their own core. Since core 1 deploys only BSW, there are no assigned data. The results of applying this allocation to the engine ECU and analyzing the improvement effect are the same as those shown in Table 4. The memory capacity is as follows. {DSPR0 = 10% (shared data: 1821, non-shared data: 11295), DSPR2 = 12% (shared data: 1709, non-shared data: 14031)}. For example, a 5ms task has an Average CET of 64.684us, which is calculated as an average core load (= Average CET/Period => 64.684us/5000us) of 0.012936 (1.2936%). In comparison, our memory allocation algorithm produced an average core load of 1.2565%, an improvement in task performance by reducing the load by 0.0371%. The access time of all tasks was reduced compared with those of the existing method, as were the average CET and average core load. The total load of all cores decreased by 2.406% (0.2216 + 0.099 + 0.0371 + 1.5372 + 0.0798 + 0.0107 +

TABLE 4. Average core load improvement of the memory allocation result.

Task	Conventional method		Proposed method		Improvement(Proposed-Conventional)	
	Avg. core load	Avg. CET	Avg. core load	Avg. CET	Avg. core load	Avg. CET
1ms	3.1827%	31.827μs	2.9611%	29.611μs	- 0.2216%	- 2.216μs
2ms	1.5907%	31.815μs	1.4917%	29.834μs	- 0.0990%	- 1.981μs
5ms	1.2936%	64.684μs	1.2565%	62.825μs	- 0.0371%	- 1.859μs
10ms	33.0084%	3300.84μs	31.4712%	3147.122μs	- 1.5372%	- 153.718μs
20ms	4.6308%	926.17μs	4.5510%	910.204μs	- 0.0798%	- 15.966μs
50ms	0.3402%	170.115μs	0.3295%	164.759μs	- 0.0107%	- 5.356μs
100ms	2.8534%	2853.45μs	2.7736%	2773.659μs	- 0.0798%	- 79.791μs
200ms	3.4499%	6899.85μs	3.3486%	6697.296μs	- 0.1013%	- 202.554μs
1000ms	0.7953%	7953.541μs	0.7847%	7847.354μs	- 0.0106%	- 106.187μs
knk	0.9359%	61.925μs	0.8481%	55.405μs	- 0.0878%	- 6.520μs
SyncS0	7.5429%	362.498μs	7.4140%	357.602μs	- 0.1289%	- 4.896μs
SyncS1	6.2957%	315.142μs	6.2835%	314.177μs	- 0.0122%	- 0.965μs

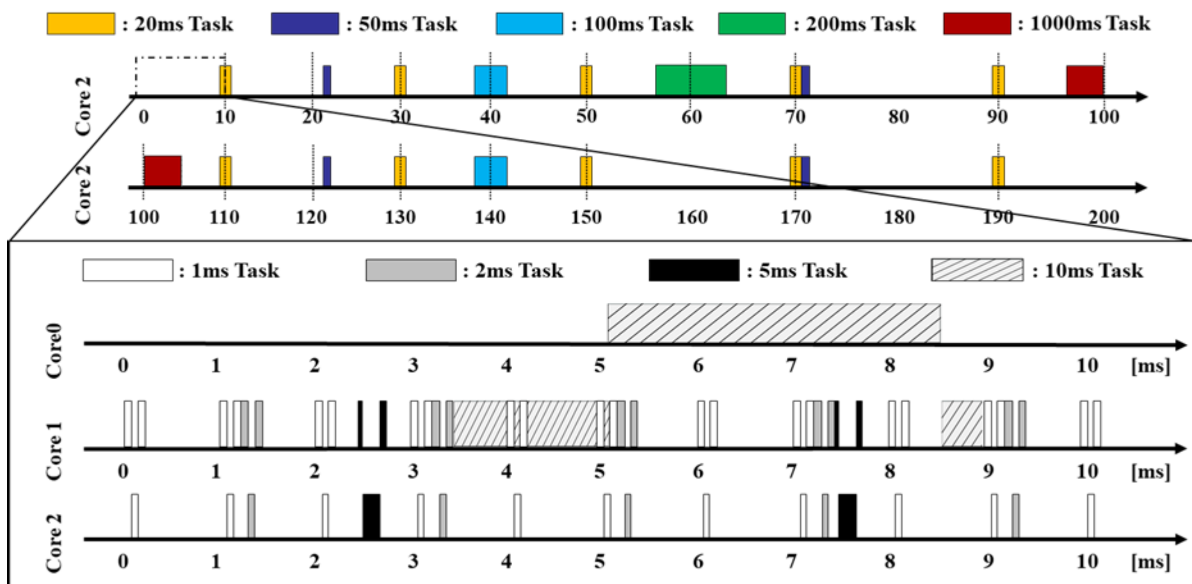

FIGURE 6. Offset allocation result of all the periodic tasks (1ms, 2ms, 5ms, 10ms, 20ms, 50ms, 100ms, 200ms, 1000ms).

0.0798 + 0.0107 + 0.0798 + 0.1013 + 0.0106 + 0.0878 + 0.1289 + 0.012 2 = 2.406). To optimize task performance,

we also implemented offset optimization technology. The offset results using above memory allocation results are shown in Fig. 6. Since the offset is concluded using the CET of each task, we divided the CET into offset avg and offset max. Offset avg is the offset value assuming the CET of all tasks will operate as an average value. In a similar way, Offset max is the offset value considering the worst core execution time (WCET). Compared with the conventional method, we set the offset to 0, which is the situation where there is no offset for any tasks. The results of our offset technology are shown in Fig. 7 and Fig. 8. When applying our offset technology, all the scheduling performances are increased compared with those at offset 0. Further, it is more efficient to use the offset avg than offset max, as shown in Fig. 7 and Fig. 8.

After adapting this optimization technology, we verified the schedulability of our multi-core-based engine ECU. HILs transmit message frames of all the ECUs in an automotive vehicle to our engine ECU. In [37], [38], h bound schedulability is pessimistic in rate monotonic analysis. Also, it can be lower than the actual achievable bound given a specific system, and does not fit our concept. (A1: All processors are allocated to single CPUs based on period, A3: There is no data dependence among processes, A4: The execution time for each process is constant) [39]. To reliably perform schedulability, we verified with the exact schedulability method [40], [41] and determined it as the most accurate and reliable method for rate monotonic verification. The real-time exact schedulability is calculated in Equation 11.

$$CET_i + \sum_{j=1}^{i-1} [\frac{t'}{P_j}] CET_j \leq T \quad (11)$$

TABLE 5. Hils results using our migration technology.

Task	Worst case *CET*	Deadline (*T*)	Exact schedulability method in Worst Case	Deadline exceed in Real-time exact schedulability	Deadlock or Malfunction error Occur
1ms	0.037523ms	1ms	0.037523 ≤ 1	No	No
2ms	0.045310ms	2ms	0.120356 ≤ 2	No	No
5ms	0.084290ms	5ms	0.407835 ≤ 5	No	No
10ms	3.394527ms	10ms	3.394527 ≤ 10	No	No
20ms	1.275906ms	20ms	2.816626 ≤ 20	No	No
50ms	0.984562ms	50ms	8.66408 ≤ 50	No	No
100ms	3.173148ms	100ms	19.225402 ≤ 100	No	No
200ms	7.512311ms	200ms	45.963115 ≤ 200	No	No
1000ms	8.678912ms	1000ms	238.494487 ≤ 1000	No	No

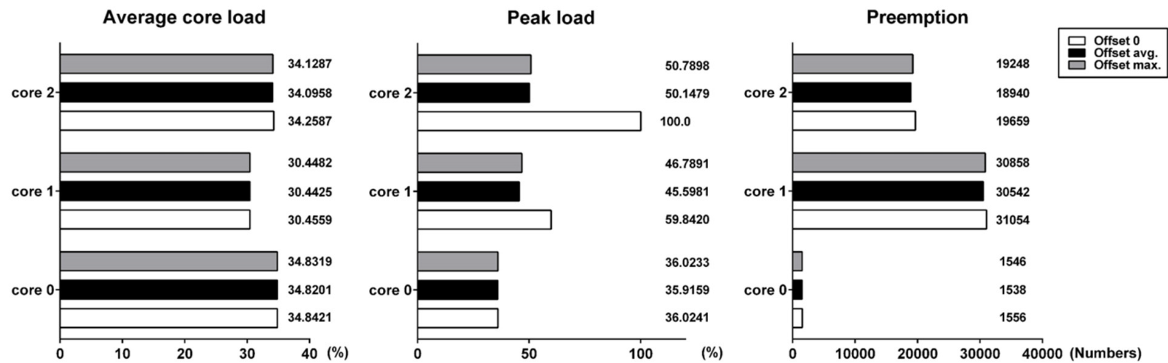

FIGURE 7. Measurement results of the Offset 0, avg, max. Average core load (Left), Peak load (Middle), Number of preemptions (Right).

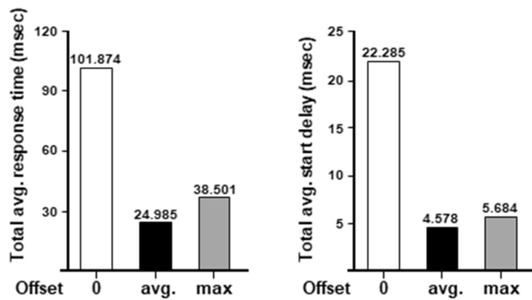

FIGURE 8. Measurement results of the Offset 0, avg, and max. Total average response time (Left), Total average start delay (Right).

- CET_i = CET of Task i
- t' = period of of Task i
- P_j = period of of Task j
- T = Deadline of Task i

The CET of each task was analyzed through timestamps at the beginning and end of tasks. The calculated CET was substituted into the exact schedulability formula to verify deadline meet in real-time. During HILs validation, no tasks exceeded the range of exact schedulability. In addition, the worst case exact schedulability was verified by substituting the measured worst case CET into the exact schedulability equation. The results are described in Table 5. Even in the worst case and real-time case, tasks did not exceed the range of deadline. All tasks were performed on schedule, and the engine ECU operated normally without malfunction or erroneous information.

IV. CONCLUSION

In this paper, we identify the key migration methodology necessary for an actual multi-core-based engine. Our migration technology includes core load balancing, shared data inconsistency, and optimization using memory and offset allocation. We apply it to the HYUNDAI engine ECU. Through HILs, the performance of tasks increases, and all the tasks work without missing deadlines. As a result, we showed that a multi-core-based engine ECU using our migration technology can be applied in an actual in-vehicle. We also confirmed the performance improvement. In future work, we will integrate the engine and transmission ECUs into a multi-core MCU through our migration method. We will also implement new prevention technology to thwart shared resource invasions and failure propagation. Using this integration technology, the number of ECUs will be reduced, leading to a reduction in power.

REFERENCES

[1] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs—Combining runnable sequencing with task scheduling," *IEEE Trans. Ind. Electron.*, vol. 59, no. 10, pp. 3934–3942, Oct. 2012.

[2] S. Schliecker, M. Negrean, and R. Ernst, "Response time analysis on multicore ECUs with shared resources," *IEEE Trans. Ind. Informat.*, vol. 5, no. 4, pp. 402–413, Nov. 2009.

[3] S. Widlund, "Migrating a single-core AUTOSAR application to a multi-core platform: Challenges, strategies and recommendations," M.S. thesis, Dept. Comput. Sci. Eng., Univ. Gothenburg, Gothenburg, Sweden, 2017.

[4] M. Broy, "Challenges in automotive software engineering," in *Proc. 28th Int. Conf. Softw. Eng.*, May 2006, pp. 33–42.

[5] *Private Communication*, HYUNDAI MOTOR PowerTrain Team, Hwaseong-si, South Korea, May 2020.
[6] H. Zeng and M. D. Natale, "Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms," in *Proc. 6th IEEE Int. Symp. Ind. Embedded Syst.*, Jun. 2011, pp. 140–149.
[7] G. Han, H. Zeng, M. D. Natale, X. Liu, and W. Dou, "Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms," *IEEE Trans. Ind. Informat.*, vol. 10, no. 2, pp. 903–908, Nov. 2013.
[8] L. Michel, T. Flaemig, D. Claraz, and R. Mader, "Shared SW development in multi-core automotive context," in *Proc. 8th Eur. Congr. Embedded Real Time Softw. Syst. (ERTS)*, Jan. 2016, pp. 547–558.
[9] X. Yao, P. Geng, and X. Du, "A task scheduling algorithm for multi-core processors," in *Proc. Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, Dec. 2013, pp. 259–264.
[10] *AUTOSAR-Specification of OS Architecture Final Version R4.2.2*. Accessed: Feb. 24, 2021. [Online]. Available: http://www.autosar.org
[11] *AUTOSAR-Specification of BSW Scheduler Final Version R1.1.1*. Accessed: Feb. 24, 2021. [Online]. Available: http://www.autosar.org
[12] M. Panic, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, and F. J. Cazorla, "RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2014, pp. 1–10.
[13] H. Salamy and J. Ramanujam, "An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 5, pp. 717–725, May 2012.
[14] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35–78, Oct. 2011.
[15] Y. Oh and S. H. Son, "Allocating fixed-priority periodic tasks on multiprocessor systems," *Real-Time Syst.*, vol. 9, no. 3, pp. 207–239, Nov. 1995.
[16] R. Hottger, L. Krawczyk, and B. Igel, "Model-based automotive partitioning and mapping for embedded multicore systems," *Int. J. Comput., Electr. Autom., Control Inf. Eng.*, vol. 9, no. 1, pp. 268–274, Jan. 2015.
[17] G. Macher, A. Holler, E. Armengaud, and C. Kreiner, "Automotive embedded software: Migration challenges to multi-core computing platforms," in *Proc. IEEE 13th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2015, pp. 1386–1393.
[18] M. Grenier, L. Havet, and N. Navet, "Pushing the limits of CAN-scheduling frames with offsets provides a major performance boost," in *Proc. 4th Eur. Congr. Embedded Real Time Softw. (ERTS)*, Jan. 2008, pp. 1–9.
[19] *Labcar Manual*, ETAS Embedded Solution, Stuttgart, Germany, 1994.
[20] *Tasking Manual*, Altium Softw. Des., San Diego, CA, USA, 1985.
[21] *ASCET Manual*, ETAS Embedded Solution, Stuttgart, Germany, 1994.
[22] *MISRA-C-2004 Manual*, MISRA, Warwickshire, U.K., 1991.
[23] M. Copic, R. Leupers, and G. Ascheid, "Efficient sporadic task handling in parallel AUTOSAR applications using runnable migration," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, Jan. 2019, pp. 603–608.
[24] *AUTOSAR-Specification of RTE Final Version R4.2.2*. Accessed: Feb. 24, 2021. [Online]. Available: http://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_RTE.pdf
[25] H. Kim, S. Wang, and R. Rajkumar, "VMPCP: A synchronization framework for multi-core virtual machines," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2014, pp. 86–95.
[26] *Association for Standardization of Automation and Measuring Systems*, MDX Manual, Hoehenkrchen, Germany, 1998.
[27] S. Miyoshi, T. Sasaki, Y. Fukazawa, and T. Kondo, "An architectural framework of snoopy interconnection for heterogeneous cache systems," in *Proc. 3rd Int. Symp. Comput. Netw. (CANDAR)*, Dec. 2015, pp. 561–565.
[28] D. Y. Kim, J. Y. Moon, and J. W. Jeon, "Time offsets for scheduling tasks in multicore ECUs," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia (ICCE-Asia)*, Nov. 2020, pp. 1–4.
[29] A. Göbel and D. Claraz, "A multi-core basic software as key enabler of application software distribution," in *Proc. Embedded Real Time Softw. Syst. (ERTS)*, Jan. 2014, pp. 1–11.
[30] *TC275 Data Manual*, Infineon Semicond. Products, Neubiberg, Germany, 1999.
[31] M. Becker, N. Khalilzad, R. J. Bril, and T. Nolte, "Extended support for limited preemption fixed priority scheduling for OSEK/AUTOSAR-compliant operating systems," in *Proc. 10th IEEE Int. Symp. Ind. Embedded Syst. (SIES)*, Jun. 2015, pp. 1–11.
[32] A. A. Bertossi and A. Fusiello, "Rate-monotonic scheduling for hard-real-time systems," *Eur. J. Oper. Res.*, vol. 96, no. 3, pp. 429–443, Feb. 1997.
[33] *INCA Manual*, ETAS Embedded Solution, Stuttgart, Germany, 1994.
[34] *Timing 1 Manual*, Gliwa Embedded Syst., Weilheim, Germany, 2003.
[35] *APP4MC Information Documentation Homepage*. Accessed: Feb. 25, 2021. [Online]. Available: https://www.eclipse.org/app4mc/documentation
[36] *TA Tool Suite Manual*, Vector GmbH, Stuttgart, Germany, 1988.
[37] N. Min-Allah, H. Hussain, S. U. Khan, and A. Y. Zomaya, "Power efficient rate monotonic scheduling for multi-core systems," *J. Parallel Distrib. Comput.*, vol. 72, no. 1, pp. 48–57, Jan. 2012.
[38] W.-C. Lu, H.-W. Wei, and K.-J. Lin, "Rate monotonic schedulability conditions using relative period ratios," in *Proc. 12th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2006, pp. 3–9.
[39] R. Dashora, H. P. Bajaj, A. Dube, and M. Narayanamoorthy, "ParaRMS algorithm: A parallel implementation of rate monotonic scheduling algorithm using OpenMP," in *Proc. Int. Conf. Adv. Electr. Eng. (ICAEE)*, Jan. 2014, pp. 1–6.
[40] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. Real-Time Syst. Symp.*, 1989, pp. 166–171.
[41] M. Park and H. Park, "An efficient test method for rate monotonic schedulability," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1309–1315, May 2014.

JUN YOUNG MOON received the B.S. degree from Sungkyunkwan University, Suwon, South Korea, in 2014, where he is currently pursuing the Integrated Ph.D. degree in electronic, electrical, and computer engineering. His research interests include multi-core programming, in-vehicle networks, automotive open system architecture (AUTOSAR), and real-time embedded systems.

DO YEON KIM received the B.S. degree from Korea University, Sejong, South Korea, in 2017, where she is currently pursuing the Integrated Ph.D. degree in electronic, electrical, and computer engineering. Her research interests include multi-core programming, automotive gateway, and real-time embedded systems.

JIN HO KIM received the B.S., M.S., and Ph.D. degrees in electronic, electrical and computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2007, 2009, and 2015, respectively.

From 2015 to 2019, he was a Senior Researcher with Hyundai Motors, Hwaseong, South Korea. Since 2019, he has been with Kyungnam University, Changwon, where he is currently an Assistant Professor with the School of Computer Engineering. His research interests include in-vehicle networks, real-time Ethernet, AUTOSAR, and real-time embedded systems.

JAE WOOK JEON (Senior Member, IEEE) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, South Korea, in 1984 and 1986, respectively, and the Ph.D. degree in electrical engineering from Purdue University, West Lafayette, IN, USA, in 1990. From 1990 to 1994, he was a Senior Researcher with Samsung Electronics Company Ltd., Suwon, South Korea. Since 1994, he has been with Sungkyunkwan University, Suwon, where he was first an Assistant Professor with the School of Electrical and Computer Engineering. He is currently a Professor with the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include robotics, embedded systems, and factory automation.