ModelMap: A Model-based Multi-domain Application Framework for Centralized Automotive Systems

Soham Sinha
soham1@bu.edu
Boston University
Boston, MA, USA

Anam Farrukh
afarrukh@bu.edu
Boston University
Boston, MA, USA

Richard West
richwest@bu.edu
Boston University
Boston, MA, USA

ABSTRACT
This paper presents ModelMap, a model-based multi-domain application development framework for DriveOS, our in-house centralized vehicle management software system. DriveOS runs on multicore x86 machines and uses hardware virtualization to host isolated RTOS and Linux guest OS sandboxes. In this work, we design Simulink interfaces for model-based vehicle control function development across multiple sandboxed domains in DriveOS. ModelMap provides abstractions to: (1) automatically generate periodic tasks bound to threads in different OS domains, (2) establish cross-domain synchronous and asynchronous communication interfaces, and (3) handle USB-based CAN I/O in Simulink. We introduce the concept of a nested binary, for the deployment of ELF binary executable code in different sandboxed domains. We demonstrate ModelMap using a combination of synthetic benchmarks, and experiments with Simulink models of a CAN Gateway and HVAC service running on an electric car. ModelMap eases the development of applications, which are shown to achieve industry-target performance using a multicore hardware platform in DriveOS.

CCS CONCEPTS
• Software and its engineering → Integrated and visual development environments.

KEYWORDS
multi-domain Simulink models, vehicle control software

ACM Reference Format:

1 INTRODUCTION
Automotive systems are continuing to increase in complexity, with the advent of advanced driver assistance systems and connected services. This has led to a corresponding increase in electronic control units (ECUs), with a modern luxury vehicle having over 100 such units [42]. An alternative approach to using large numbers of separate ECUs is to develop a centralized vehicle management system (VMS) [6] where functions are consolidated as software tasks on a single multicore machine [55]. Software is more easily upgraded and extended, without the cost of added electronics [23]. However, software tasks must now implement functions with different timing, safety and security requirements on the same hardware.

To address this, VMSs assign tasks having a high criticality, or consequence of failure, to a real-time OS (RTOS), while low criticality tasks run on a general purpose OS (GPOS) [1, 3, 7, 24, 41, 55]. Example high criticality tasks are those that affect vehicle control, including steering, throttle and braking, while low criticality tasks include infotainment services. Both the RTOS and GPOS are able to run on a single hardware platform using machine virtualization.

A significant challenge is how to develop, deploy and support communication between mixed-criticality tasks running in different domains, or operating systems, on multicore VMSs. To date, most vehicle functions such as heating, ventilation and air conditioning (HVAC) or powertrain control are developed for simple, single-core ECUs. These ECUs host a single, simple RTOS or firmware. Engineers accustomed to model-based design languages such as Simulink [21] and LabView develop functions for these ECUs without awareness of control flow (e.g., threads), data structures, and low-level communication primitives. Model-based design languages have thus far lacked support for multi-OS domain systems, leaving the burden on expert programmers to port ECU functions.

We introduce ModelMap, a model-based multi-domain application development framework for automotive functions in our in-house centralized VMS, called DriveOS. DriveOS leverages a partitioning hypervisor to host RTOS and Linux sandboxed domains on top of a multicore PC-class x86 machine [55]. ModelMap implements a set of Simulink interfaces that target multiple sandboxes, or OS-level protection domains, in DriveOS.

ModelMap supports binding a real-time periodic thread to a Simulink control task for timing-predictable execution. It provides synchronous and asynchronous inter-task communication primitives, and real-time I/O for commonly used protocols such as controller area network (CAN) bus. Vehicle functions that span OS domains are encapsulated as nested binaries, which support the deployment of executable code for multiple application binary interfaces. To the best of our knowledge, ModelMap is the first open model-based multi-domain VMS application framework.

We present two Simulink automotive software models and implement them using ModelMap Simulink blocks in DriveOS. These models are: (1) a CAN Gateway service, which delivers CAN messages to different software threads in DriveOS, according to end-to-end timing guarantees, and (2) a port of an HVAC controller for
an electric vehicle being developed with our partner company. We demonstrate the HVAC model’s functional and timing correctness with a model-in-the-loop (MIL) and hardware-in-the-loop (HIL) execution equivalence against real-world data traces.

The contributions of this paper are three-fold: (1) we introduce the first model-based multi-domain application development and deployment framework for a VMS; (2) we demonstrate that Simulink models running on a multicore x86-based centralized VMS have predictable end-to-end delays; (3) we illustrate a Simulink model’s functional and timing correctness with MIL and HIL equivalence.

The next section provides motivation and background to model-based design in DriveOS. Section 3 describes the ModelMap tools for multi-domain code development. Section 4 explains the nested binary concept in the DriveOS VMS. An evaluation of ModelMap is covered in Section 5, using simulated and real-world data sets as well as CAN Gateway and HVAC case studies. This is followed by related work and conclusions, respectively, in Sections 6 and 7.

2 MOTIVATION AND BACKGROUND

Simulink Model-based Design. MATLAB/Simulink is the de-facto design tool for model-based vehicle control software. Proprietary functional blocks are commonly provided by vendors for use on their own ECUs, which traditionally feature single-core microcontrollers. However, the growing popularity of embedded multicore processors has led to the development of task-parallel Simulink models [5, 44, 45]. Notwithstanding, there appears to be a lack of Simulink model-based design frameworks for multi-OS domain systems. This has consequently motivated our work, which targets the use of Simulink development tools in hypervisor-based [13, 50, 58] vehicle management systems.

In Simulink, a control task is modeled with a number of functional blocks that are connected with signals. Each functional block derives output signals from a combination of its inputs and internal states. A subsystem comprises one or more functional blocks. A complete control system is developed from multiple subsystems. In a modern VMS, these subsystems may require execution in different OS domains, depending on criticality and functionality. For example, a CAN Gateway [55] might run within an RTOS to directly access a timing- and security-critical vehicle bus for chassis and powertrain control. At the same time it may wish to transfer processed data from the CAN bus via secure shared memory to a Linux domain, to update a less critical instrument cluster service that relies on a graphical display. Likewise, a Linux domain user-interface to control cabin temperatures might interact with an HVAC controller within an RTOS. ModelMap eases the development of applications for these types of multi-domain systems.

Centralized VMS. A centralized VMS consolidates multiple ECU functions onto a multicore platform. Drako Motors’ DriveOS™ [54, 55], Mercedes-Benz’ MB.OS [41] and Toyota’s Arene OS [1] are examples of centralized VMSs. These all support multiple guest operating systems, or sandbox domains, running on a partitioning hypervisor [46, 58]. They provide temporal and spatial isolation between guest sandboxes so that tasks executing within one sandbox do not interfere with tasks in other sandboxes. The hypervisor is removed from normal sandbox execution, allowing guest operating systems to execute directly on their assigned hardware.

This paper addresses the tools and mechanisms for multi-domain functional development and deployment, in the context of DriveOS, as shown in Figure 1. A Quest real-time OS [14] domain works in unison with a general-purpose Linux OS domain. For proof of concept, this paper focuses on two single-core domains, although DriveOS is capable of supporting more domains and cores.

3 ModelMap DESIGN TOOLS

Figure 2 shows an overview of the ModelMap code generation for a multi-domain application. A model is first designed with ModelMap and other Simulink blocks. Then, ModelMap block-level and DriveOS system Target Language Compiler (TLC) [53] files are utilized by the Simulink Embedded Coder, to generate the domain OS-specific C source code. OS-specific versions of gcc cross-compile C code into Quest and Linux ELF binaries. Finally, a nested binary compiler (see Section 4) creates a multi-domain binary executable.

A custom Embedded Real-Time (ERT) system TLC file [53] specifies the C code generation from Simulink model blocks. This is explained further in Section 3.1.4.

3.1 Thread Setup Blocks

A threadSetup Simulink block is used to create periodic threads for either Quest or Linux, and aperiodic threads restricted to Linux. The block details are summarized below:

- **Block Type:** C MEX S-function [34].
- **Block Parameters:** A Simulink block mask [35] identifies thread-specific parameters. These include the Thread Name and Domain OS (Quest or Linux). A Quest periodic thread is further parameterized with a Runtime and Period. A Linux domain periodic thread has a Runtime, Period and Deadline, while a Linux-only aperiodic thread has no further parameters.

Figure 1: High-level Design of DriveOS

Figure 2: The ModelMap Workflow
3.1.1 Quest Periodic Threads. Timing and safety critical control tasks run as periodic threads in Quest. Application threads in this domain leverage Quest-specific and C library APIs, to perform real-time I/O and secure CAN bus operations.

Example tasks include HVAC and powertrain control, which are designed as Simulink function-call subsystems. The function trigger ports of these subsystems are connected to the output port of a corresponding threadSetup block. threadSetup is configured with a Thread Name parameter, and the Domain is set to Quest. A model developer provides the Runtime and Period parameters that are respectively set as the budget (C) and period (T) of a Quest periodic task. \( \tau \) receives its budget, C, every period, T, when runnable.

3.1.2 Linux Periodic Threads. Linux is the lower criticality GPOS domain in DriveOS. Although not a hard real-time OS, it provides sufficient timing critical guarantees for SCHED_DEADLINE [27] using the PREEMPT_RT patch [47]. Linux provides complementary support for Quest with its libraries, device drivers and services that would take many years of development to implement in a newRTOS. A developer provides the Runtime, Period, and Deadline threadSetup block parameters that are passed on to the SCHED_DEADLINE policy via the Linux sched_setattr system call.

3.1.3 Linux Aperiodic Threads. A centralized VMS also runs non-timing critical operations such as logging, storage and over-the-air updates. threadSetup supports these tasks as Linux pthreads. No additional threadSetup block parameters are needed.

3.1.4 Code Generation. The threadSetup S-function block’s properties are described in a C MEX (threadSetup.c) file for simulation, and in a TLC file (threadSetup.tlc) for code generation. As the threadSetup block is designed for model deployment in a DriveOS system, threadSetup.c only saves the block parameters for code generation, without any simulation. In the threadSetup.tlc file, three key steps are followed to generate its corresponding C code [32].

1) The BlockInstanceSetup TLC function of an S-function block is executed at the very start of code generation. ModelMap uses this function to retrieve all the block parameters from the simulation environment.

2) The Start TLC function (Code Block 1) includes any initializing code in the final C source code. ModelMap provides this function to assign the block parameters (Runtime, Period, and, if applicable, Deadline) to a DriveOS C structure (driveos_sched_param_t) with domain-specific members.

```c
#ifdef QUEST
for(i = 0; i < num_threads; i++) {
    #ifdef QUEST
        pthread_t* new_thread = (pthread_t *) malloc(sizeof(pthread_t));
        pthread_create(new_thread, NULL, all_thrdParms[i].threadfuncname, all_thrd_Parms[i]);
    #else
        // Linux domain
        #endif
}
#endif
```

Code Block 2: A snippet of the ModelMap-generated main function in C

3.2 Inter-task Communication Blocks

ModelMap provides a set of blocks for intra- and inter-domain task communications. These blocks are set up as shared memory communication channels by the DriveOS hypervisor [55]. Intel VT-x extended page tables (EPTs) securely map host physical memory regions between communicating threads, irrespective of their domain. Both synchronous and asynchronous communications are supported across channels identified with a unique channel1_key.

DriveOS has a set of C API functions in Linux and Quest to set up the communication channels. ModelMap implements MATLAB interfaces for these C functions [33], which are described below.

A ModelMap createChannel Simulink block takes an integer input as the channel1_key. It has block mask parameters to set the type and specification of the channel. For a synchronous channel, the buffer length and the size of each element must be specified. For an asynchronous channel, only the element size is needed.

3.2.1 Synchronous Communication. A synchronous channel is implemented as a ring buffer in DriveOS. This is useful for control data that must be communicated without loss. Channel data structures are created by OS-specific uspares libraries in both Quest and Linux domains. syncRead and syncWrite are busy-waiting calls that read and write, respectively, a message in the buffer. Busy-waiting is used for synchronous communication in Quest-V [58] and DriveOS [55].

Blocking or busy-waiting is problematic when reading or writing to different channels. As will be seen later in Figure 10, a syncRead on one channel may delay the execution of syncRead calls on other channels. To mitigate this issue, ModelMap implements syncRWRead and syncRNWrite, which are non-waiting (NW) functions. Calling syncRWRead (or syncRNWrite) when a channel buffer is empty (or full) immediately returns -1, otherwise it returns the size of the read (or written) message.
3.2.2 Asynchronous Communication. An asynchronous channel in DriveOS is implemented using Simpson’s four-slot protocol [52]. This is useful when the most recent (i.e., freshest) data must be communicated, while stale data is discarded. Sensor readings fall into this category of communication. The asyncRead and asyncWrite Simulink blocks are available to read and write asynchronous messages, respectively.

3.3 CAN I/O Blocks

CAN communication between sensors and actuators is commonly used in the automotive domain. DriveOS implements real-time USB-CAN I/O in its Quest domain. The Quest CAN I/O API is accessed via MATLAB’s C-interface function blocks, including canChannelSetup, canRead, and canWrite. canChannelSetup has a block parameter to set the CAN baud rate from 10 kbit/s to 1Mbit/s.

3.4 Timing Blocks

ModelMap implements several Simulink timing blocks. MATLAB function block time_from_start outputs the time in µs since the model starts running. Similarly, time_since_last_called outputs the time in µs since the last time this block was called. These timing functions use the x86 RDTSCL instruction to measure processor clock cycles, divided by the base clock frequency, to yield accurate time in µs. These blocks are C-function interfaces in MATLAB [33].

3.5 Domain-specific C Code Generation

Domain-specific components of a model are designed as subsystems in Simulink. The sibuild(<subsystem name>) command is used by ModelMap to generate domain-specific C code for a target Simulink subsystem.

4 NESTED BINARIES

ModelMap produces an ELF binary [31] format for DriveOS, called a nested binary. This is similar to a fat binary [17], and contains multiple executables with Application Binary Interfaces (ABIs) for different operating systems. For example, DriveOS nested binaries contain executable images for both Quest and Yocto Linux.

ModelMap includes a nested binary compiler. A corresponding nested binary loader performs runtime parsing of individual executables within a nested binary. It then spawns a new process for every executable into a corresponding DriveOS sandboxed domain.

4.1 Nested Binary Format

As shown in Figure 3, a nested binary section header table contains file offsets to $N \geq 1$ binexec sections. Each binexec section stores the ELF binary data for a specific domain executable. A nested binary metadata ELF section also stores the mapping between an individual ELF binary and a runtime domain ID.

4.1.1 ELF Header. The ELF header defines the target OS ABI, bitness, and other details in an ELF binary. The following fields are modified: (1) e_ident[EI_OSABI]: The target OS ABI is set to a custom value of 0x15; (2) e_machine: The target ISA is set to EM_386 (0x386) for an x86 target; (3) e_type: The object file type is set to 0x82 for an executable file; (4) e_ident[EI_DATA]: The endianness is set to little-endian.

4.1.2 Program Header Table. This section has one entry to satisfy the ELF format requirement. The entry is the Program Header Table (PT_PHDR) itself. As individual executables have their own program header information, this section is not needed.

4.1.3 Section Header Table. This section lists all the data sections in a nested binary:

- `binexec`: Every nested binary has a separate binexec section for each ELF binary executable. The name of every binexec section is appended at the end with an integer numeric ID, starting from 1. This ID specifies the order in which the binaries will be spawned at runtime, where lower ID means earlier execution.
- `metadata`: The metadata section maps an individual binary in a binexec section to a domain in DriveOS. The nested binary loader uses the metadata section to spawn a new process from an individual binary in the corresponding domain. The section contains an array of C structs which holds a tuple of the binexec section name and the corresponding integer domain ID. Currently, DriveOS assigns domain ID 1 to Quest and 2 to Linux.
- `shstrtab`: This is the string table section that contains the section names, like other ELF binaries.

4.2 Nested Binary Compiler

ModelMap’s nested binary compiler (nested_bin_cc) creates a nested ELF binary from multiple individual binary executable files. The compiler utilizes the libelf library [25] to create, enumerate and organize different ELF binary sections according to the above format. The following command is used to create a nested binary:

```
<Binary File1> <Domain ID1> ... <Binary FileN> <Domain IDN> <Name of Nested Binary>
```

The above command combines N ELF binary executable files and saves the mapping between a binary and its runtime domain in the metadata section. For example, Binary File1 is mapped to Domain ID1. A new nested binary is created with the name specified in the last argument. binutils tools such as readelf support inspection of nested binary sections.

4.3 Nested Binary Loader

The nested binary loader is also implemented with the libelf APIs [25]. The loader runs in a DriveOS Linux domain and takes a nested binary as the first argument. It also takes a number of command-line arguments for every individual ELF binary. The following command is used to execute a nested binary:

```
nested_loader <nested ELF binary> <argc1> <argv11> ... <argc2> <argv21> ...
```

Figure 3: Nested Binary Sections
Here, argc1 is the number of command-line arguments for the binexec1, starting with argv11. Similarly, argc2 is the argument count for binexec2, starting with argv21, and so on.

The loader parses the metadata section in a nested binary to read the mapping between a binexec section and its runtime domain. It spawns a new process with the raw bytes of a binary embedded in a binexec section to its respective domain. Figure 4 summarizes the steps to execute a binary in Linux and Quest.

4.3.1 Executing a binary in a Linux Domain. The nested binary loader parses the metadata section of a target binary to locate the corresponding binexec sections for Linux (Domain ID=2). The loader employs the fork-and-execute approach in Linux to spawn the executable image as a new process. However, as the executable is not in a file and directly available in memory, the execve-class of C functions cannot be used. Instead, the loader creates a new file descriptor for the memory location of the Linux binary with the memfd_create function [29]. Then, it forks a new child process and calls fexecve [28] to start execution.

4.3.2 Executing a binary in a Quest Domain. In this case, the Linux nested binary loader first identifies the domain ID=1 for a Quest binary object. It then sends the executable bytes to Quest via shared memory as shown in Figure 4. A specific shared memory region of 800KB is mapped between the Quest and Linux domains at system boot time for remote binary execution. The region is appropriately sized to accommodate typical Quest static binaries. This shared memory region works as a synchronous ring buffer channel with a single buffer slot. A remote binary loader process in Quest polls the shared region for any new binary execution request. Once the Linux nested binary loader indicates that it has written a new program to the shared region, the remote binary loader in Quest starts reading the program and its arguments. Then, the remote Quest loader spawns a new process with a fork-and-exec mechanism.

5 EVALUATION

In this section, ModelMap is tested with custom and real-world Simulink models. The goal is to show that the end-to-end (E2E) delays, or the maximum reaction times [15], of the models are within their expected upper bounds after deployment in DriveOS.

ModelMap Simulink blocks are applied to the following three models: (1) a multi-domain synthetic benchmark with different types of inter-task communication, (2) a DriveOS CAN Gateway [56] to filter and forward CAN messages to different Quest and Yocto Linux applications, and (3) a port of an automotive HVAC Simulink model for a MotoHawk ECU to the DriveOS VMS. The functional and timing correctness of the ported HVAC model is demonstrated by the output equivalence in both MIL and HIL execution.

The three models are tested with the DriveOS VMS running on a Cincoze DX1100 industrial PC [12], featuring an Intel 2.4GHz i7-8700T processor. The DX1100 is connected to a Kvaser Pro 5xHS USB-CAN adapter [26].

DriveOS supports real-time USB-based CAN I/O using Quest. Quest handles device interrupts in the context of time-budgeted, schedulable threads [14]. The test setup uses two USB xHCI bottom-half handler threads, each with budget=0.1ms and period=1ms, referred to as USBBH_rx and USBBH_tx. Two USB-CAN kernel device threads, each with budget=0.2ms and period=1ms, send (CAN_rx) and receive (CAN_tx) CAN messages.

5.1 Synthetic Benchmarks

Figure 5 shows our multi-domain Simulink benchmark model, designed using ModelMap blocks. The inter-task and CAN channel setup blocks are not shown for space constraints. The model reads a CAN message in the canReader subsystem of the Quest domain (Domain 1) from CAN channel 0 (CAN0) via canRead. Then, synWrite forwards the data to the procThread subsystem for processing in the Linux domain (Domain 2). This setup allows procThread to apply any control logic to the received message using additional Simulink blocks. For our experiments, procThread forwards the message to a canWriter subsystem in Domain 1 using synWrite. The canWriter then outputs a message on CAN channel 1 (CAN1). This model is representative of the canonical communication path between two CAN bus interfaces and separate OS domains in DriveOS.

The canReader, canWriter and procThread blocks in Figure 5 are function-call subsystems, configured as periodic threads. Their function trigger ports are connected to the output ports of the threadSetup blocks. The threadSetup blocks are assigned to the Quest domain for the canReader and canWriter subsystems, and to the Linux domain for the procThread subsystem. Their budgets and periods, given in Table 1, are derived empirically by profiling [59], and assigned in the corresponding threadSetup blocks. canReader and canWriter subsystems rely on Quest real-time capabilities. procThread is representative of a lower criticality control task that is scheduled in the Linux domain using the SCHED_DEADLINE policy.

The Simulink model’s corresponding C code and subsequent nested binaries are automatically generated. When this model is launched by ModelMap’s nested binary loader, canReader and canWriter threads are spawned in Domain 1, and procThread is
5.1.1 End-to-end Delay Performance. We measure the end-to-end (E2E) delay (also known as the maximum reaction time) [11, 15, 18] of a CAN message traversing through canReader → procThread → canWriter threads. The E2E delay upper bound for a pipeline of periodic real-time tasks has been theoretically analyzed before [2, 15, 18], but only for a domain-specific scheduling algorithm. However, it is important to measure the E2E delay of multi-domain applications in a centralized VMS where the time-critical software components expand beyond a single domain [3, 55]. We perform an experimental evaluation of such multi-domain applications in this paper and use the sum of the task periods [22] (optimistic) and Davare’s upper bound [15] (conservative, twice the sum of periods assuming response-time of a task ≤ its period) as the two target E2E delay bounds. As stated earlier, the USBBH_rx, USBBH_tx, CAN_rx and CAN_tx threads are also considered in a task chain for CAN I/O, as a CAN message has to pass through these I/O threads as well. For example, the aggregate period delay bound for Figure 5 will be \((1 + 1 + 1 + 1) [\text{for the I/O system threads}] + (2 + 2 + 1) [\text{from Table 1}]\) = 9ms.

5.1.2 Result Analysis. A stream of messages is sent from an Ubuntu 18.04 Linux machine to DriveOS via CAN0 on the DX100. A corresponding message is received via CAN1 on the same Ubuntu machine. The sent and received CAN message timestamps are logged with candump in Ubuntu, to calculate the E2E delay. In the first set of experiments, we vary the utilization (ratio of Runtime and Period) of the procThread subsystem from 10 to 50% by increasing the Runtime parameter in the associated threadSetup block. Figure 6a shows the minimum, average and maximum E2E delays with increasing procThread utilization in Linux. All the E2E delays are within the target upper bounds.

Figure 6a shows that the maximum E2E delay is improved by 46%, as procThread’s utilization is increased from 10% to 50% in Linux. This coincides with the increased fraction of all E2E delays within the x-axis bound in Figure 6b. The median latency every 10 frames in Figure 7 is more variable for the 10% case than others. Allocating more utilization to a Linux domain subsystem not only improves the maximum E2E delay but also reduces jitter. However, CPU utilization is often limited in resource-constrained automotive systems. ModelMap’s maximum E2E guarantee is crucial for time-critical control software modeling.

5.1.3 procThread in RTOS vs. Linux. The next experiment compares the previous model to one where the procThread subsystem in Figure 5 is moved to Domain 1, leaving Domain 2 idle. A special BG scheduling mode in Quest is also tested. This mode gives additional CPU time to a task beyond its model-specified CPU time via background scheduling, if other tasks do not need anymore CPU.

---

Table 1: Budgets and Periods for the Synthetic Benchmark

<table>
<thead>
<tr>
<th>Subsystem/Thread</th>
<th>Budget (µs)</th>
<th>Period (µs)</th>
<th>Util. (%)</th>
<th># of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>canReader</td>
<td>100</td>
<td>2000</td>
<td>5%</td>
<td>1</td>
</tr>
<tr>
<td>canWriter</td>
<td>100</td>
<td>2000</td>
<td>5%</td>
<td>1</td>
</tr>
<tr>
<td>procThread</td>
<td>100-500</td>
<td>1000</td>
<td>10-50%</td>
<td>1</td>
</tr>
</tbody>
</table>

Domain 1

Domain 2

5.1.4 Asynchronous Communication Block. The next experiment replaces all syncRead (and syncWrite) blocks with asyncRead (and

---

Figure 6: Benchmark E2E Delay vs Domain 2 Task Utilization

Figure 9a shows the E2E delays when increasing the procThread utilization up to 30\% \(^1\), keeping its period fixed at 1ms. All E2E delays are under the target upper bounds. The maximum E2E delays for the Quest BG mode stay almost the same, as procThread leverages additional CPU time. Disabling BG mode still yields E2E delays under the target bounds, but the maximum ones are worse than Linux for higher utilization. As procThread’s period is fixed, its priority remains the same in Quest, even with higher utilization. Therefore, the maximum E2E delay does not decrease as much as it does while running in Linux, where only procThread is executed.

In another experiment, procThread’s period is increased from 1ms to 8ms, keeping its utilization fixed at 10%. The results in Figure 9b show that the E2E delays are increased with greater procThread period. If a CAN message is not handled in the same job (i.e., task instance) that it is received, it might wait for potentially more than a task’s period to be transferred. Therefore, E2E delays increase with higher procThread periods. However, the maximum E2E delays are within the target upper bounds, except for the 8ms period in Linux where it violates the aggregate period bound. As the procThread period in the 8ms case is significantly more (4x) than the periods of canReader and canWriter threads (2ms), buffering delays increase the maximum E2E delay beyond the sum of periods. Nevertheless, all the delays are well under Davare’s bound.

---

\(^1\) >30\% is not possible due to the rate-monotonic scheduling bound with other tasks.

---

Figure 7: Synthetic Benchmark

Figure 8: Loss Rate in Async Channel
asyncWrite) blocks in the model of Figure 5. In asynchronous communication, if a receiver task has a greater period than a sender task, then a message is potentially overwritten by the sender, before it is observed by the reader. The number of lost messages is important in asynchronous communications. Figure 8 shows the loss-rate (ratio of number of lost messages and total messages) against increasing procThread period, while it is run in Linux and Quest.

A stream of 1275 CAN messages are sent at 5ms intervals from the Ubuntu machine. In Figure 8, as long as receiving procThread’s period in Linux is less than the sending canReader’s period of 2ms, there is no data loss. The loss-rate increases with greater periods from 2ms. As procThread’s period goes greater than or equal to canReader’s period, procThread starts missing CAN messages. In the Quest-only model, no loss is observed until 8ms period, because the source message rate (1/5ms=200Hz) is greater than the rate of all the tasks, and they are all running with the same RMS scheduling policy. However, when procThread runs in Linux it is scheduled earliest-deadline first according to the SCHED_DEADLINE policy. As Quest tasks are scheduled in RMS order there is a potential priority mismatch, highlighting the importance of correctly setting task periods for multi-domain task models.

5.2 Case Study 1: CAN Gateway

DriveOS’s CAN Gateway is shown as a ModelMap Simulink model in Figure 10. It is used to distribute messages between different domains and CAN buses.

As before, CAN messages are read via CAN0 in Quest (Domain 1) and forwarded to Linux (Domain 2) by a canReader subsystem. A canWriter subsystem receives CAN messages from Linux to be sent out via CAN1. Unlike the previous benchmark model, there are multiple subsystems in Linux to process different categories of CAN messages based on their CAN IDs.

Figure 10 shows N Linux subsystems (forwarder{1…N}) where \( N \in \{1, 2, 4, 8\} \) in our experiments. Each forwarder subsystem is connected to two inter-task synchronous channels: one is to receive CAN messages from canReader, another is to send CAN messages to canWriter. If a different Linux application wants to receive (or send) a message of any particular CAN ID, it has to request it from the specific Linux forwarder subsystem of the CAN Gateway. For example, Instrument Cluster and In-vehicle Infotainment applications in DriveOS request CAN message transfers via specific Linux forwarder subsystems. For the E2E delay overhead of the CAN Gateway, forwarder{1…N} pass through the CAN messages from their incoming inter-task channel (from canReader) to the outgoing channel (to canWriter).

The syncNWRead block in canWriter is used to read from inter-task channels, without busy-waiting when a buffer is empty. Experiments show that syncNWRead significantly improves the E2E delay. syncNWWrite blocks are not used in canReader, as the Linux subsystems keep the inter-domain communication buffer free by reading out messages at a suitable rate.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Budget (µs)</th>
<th>Period (µs)</th>
<th>Util. (%)</th>
<th># of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>canReader</td>
<td>200</td>
<td>2000</td>
<td>10%</td>
<td>1</td>
</tr>
<tr>
<td>canWriter</td>
<td>300</td>
<td>1000</td>
<td>20%</td>
<td>1</td>
</tr>
<tr>
<td>forwarder{1–8}</td>
<td>6400–800</td>
<td>8000 (fixed)</td>
<td>80%–10%</td>
<td>1–8</td>
</tr>
</tbody>
</table>

5.2.1 Result Analysis. Table 2 shows the budgets and periods of all the CAN Gateway tasks. The E2E delays are plotted in Figure 11 against increasing numbers of Linux forwarder threads, keeping their total utilization at 80%. For example, if two forwarder threads are executed, then each of them has 40% utilization. The E2E delays in Figure 11a exhibit low jitter and remain under the target bounds in all cases. This shows that ModelMap’s CAN Gateway is able to handle multiple Linux threads in a DriveOS VMS system.

In other experiments, syncNWReads are replaced with syncRead blocks in the canWriter subsystem. Figure 11b demonstrates that the polling syncRead block increases the E2E delay sharply as the number of threads increases. syncNWRead blocks are important for a scalable CAN Gateway as the busy-waiting times on synchronous channels are prohibitively large with more threads.

5.3 Case Study 2: Automotive HVAC Control

In this study, an HVAC controller running on a MotoHawk ECU is ported to DriveOS using ModelMap Simulink blocks. The HVAC Simulink function-call subsystem is connected to a threadSetup block. threadSetup configures the HVAC subsystem to run in the Quest domain with 0.5ms budget and 5ms period. The HVAC subsystem communicates with the Linux domain to save settings in persistent storage for when the vehicle is restarted. The functional and timing correctness of the HVAC control is investigated.

The HVAC control receives input signals via 18 CAN IDs and sends the output signals via 7 additional IDs. CAN I/O is via two inter-task communication channels with the CAN Gateway mentioned above. The HVAC control avoids waiting on any CAN IDs by using syncNWWrite and syncNWRead blocks for message transfers. The CAN Gateway canReader and canWriter threads are set to 0.2ms budget and 4ms period.
The HIL outputs of the HVAC model after its deployment in DriveOS are compared with the MIL outputs in Simulink, using CAN data traces from our electric car. Every CAN input message is tagged with a unique and monotonically increasing integer Tag ID, which is passed through to the HVAC control's output CAN messages. The HIL and MIL signal values in the HVAC control are checked to see that they match for all Tag IDs. Due to space limits, the HIL and MIL Driver Temperature signal output signals are shown over 30 seconds in Figure 12. For Tag IDs 756 and 762, driv_temp is respectively 1 and 2 in both HIL and MIL simulations. This is observed for all the Tag IDs and signals. The time difference on the x-axis is the DriveOS system and HVAC control overhead and contributes to the signal reaction time. Reaction times for all the signals in the HVAC control are in the similar range of 160–180 ms, which is deemed acceptable for our vehicle.

5.4 System Overheads

ModelMap overheads are measured with a series of microbenchmarks, averaged over 20 runs. The x86 RDTSC instruction is used for timing measurements, having an overhead of 0.04μs or ~96 clock cycles, which is subtracted from all delays.

Table 3: System Overheads for the DriveOS Simulink Blocks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadSetup</td>
<td>53</td>
</tr>
<tr>
<td>channelCreate(Sender,Receiver)</td>
<td>5722</td>
</tr>
<tr>
<td>channelConnect</td>
<td>5699</td>
</tr>
<tr>
<td>(async)Read/Write</td>
<td>0.01-0.03</td>
</tr>
<tr>
<td>canChannelSetup</td>
<td>-</td>
</tr>
<tr>
<td>canRead/Write</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3 presents the overheads of the ModelMap Simulink blocks in DriveOS. The threadSetup block takes more time in Quest than it takes in Linux, because Quest has to create a sporadic server abstraction for RMS [14, 57]. The creation and connection to an inter-task communication channel can be expensive VMExit [46, 55] operations to the underlying hypervisor and take more time than reading/writing to the memory-mapped channels. CAN channel setup takes 18ms to configure the transfer rate of the USB-CAN interface. These blocks are only applied in an initial setup phase without significant runtime costs.

5.4.1 Nested Binary Measurements

A nested binary’s size is the sum of all individual binaries and 14-bytes of metadata per binary. Table 4 presents the overheads of executing a nested binary. Quest only supports static binaries for fast and predictable runtime, so they are typically larger than Linux dynamically-linked binaries.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extracting an individual binary from a nested binary</td>
<td>0.16</td>
</tr>
<tr>
<td>Forking a process via memory in Linux</td>
<td>0.11</td>
</tr>
<tr>
<td>Sending ~300KB binary from Linux to Quest</td>
<td>7.13</td>
</tr>
<tr>
<td>Receiving ~300 KB binary in Quest and forking it</td>
<td>184</td>
</tr>
</tbody>
</table>

6 RELATED WORK

Pagetti et al. have done extensive work on periodic Simulink models for multicore platforms based on the ROSACE architecture for avionics [5, 44, 45]. They employ formal verification techniques from design to code generation, to meet avionics standards. However, multicore CPU modeling and verified code generation remains a challenge, and relies on faithful code translation from other high-level formally defined design languages like Lustre.

Formal verification has been applied in the context of Simulink to C code generation [4], and is being used by automotive companies [21]. Other work has investigated design-level verification of Simulink models [9, 16, 19, 40, 48]. Fons-Albert et al. [20] applied model-based design to integrated modular avionics using the XtratuM hypervisor [13], focusing on application partitioning, automatic code generation and real-time tasking. However, little has been done on multi-domain code generation and deployment.

Emerging multi-domain vehicle management systems [1, 3, 41, 54, 55] require redesigned control function development tools [51]. Although new approaches are being considered [10, 49], they mostly target ECU-based systems [8]. In recent years, MathWorks has presented Simulink Desktop Real-time [37] for real-time simulation of models. However, it is not a VMS solution. Simulink now supports Linux and VxWorks [60] tasks, but without any periodic control mechanism [38]. ModelMap presents an end-to-end model-based framework for next-generation multi-domain VMS platforms.

7 CONCLUSIONS AND FUTURE WORK

This work presents ModelMap for model-based multi-domain application development and deployment in a centralized vehicle management system. ModelMap consists of Simulink blocks for binding real-time threads of control, inter-task communication and CAN I/O. It supports the generation of nested binary executables to encapsulate and execute DriveOS applications. Experiments show that custom and real-world DriveOS Simulink models, designed using ModelMap, have predictable end-to-end delays, in keeping with the requirements of a high performance electric vehicle.

Future work will use ModelMap to integrate additional vehicle control functions related to powertrain and battery management into DriveOS. Plans are underway to extend ModelMap interfaces with a model-driven pipeline programming language for multi-/manycore systems spanning multiple OS domains.

8 ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF) under Grant # 2007707. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. Special thanks are also given to our colleagues at Drako Motors.