VM-Based Power Management for Embedded Multicore Systems

Kazuaki Tanaka[1(✉)], Hayato Tsuru[1], Ko-ichiro Sugiyama[2], and R. Krishnamoorthy[3]

[1] Kyushu Institute of Technology, Fukuoka, Japan
kazuaki@ics.kyutech.ac.jp
[2] National Institute of Technology, Matsue College, Matsue, Japan
[3] Chennai Institute of Technology, Chennai, India

Abstract. Multi-core systems in embedded systems improve performance and enable power reduction through core-level power management. However, conventional implementations require complex low-level programming.

This paper presents a VM-based program execution model for a multi-core microcontroller. This approach simplifies multi-core and multi-task execution through a script programming environment that manages memory contention and synchronisation issues, thereby reducing application-level complexity. A prototype implementation using mruby/c on a dual-core microcontroller platform evaluates the supporting overhead for multi-core and multi-task execution.

Keywords: mruby/c · Embedded Multicore Systems · Virtual Machine (VM) · Power Management · Memory Management

1 Introduction

The Internet of Things (IoT) and edge computing applications have increased the demand for greater performance and flexibility in embedded systems. To achieve these requirements, recent microcontrollers have adopted multi-core architectures, even in resource-constrained environments. Multi-core embedded systems not only enable the parallel execution of tasks and improved throughput, but also reduce power consumption through task management, synchronisation and power-aware operation.

Traditionally, embedded multi-core systems have been programmed using low-level languages such as C, which requires developers to manage task scheduling, memory access and synchronisation between cores explicitly. Maintaining correctness in shared-memory environments is particularly challenging due to race conditions and execution state management. Consequently, developing applications for embedded multicore systems remains complex and can sometimes result in serious errors.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026 Workshops, LNCS 16761, pp. 465–478, 2027.
https://doi.org/10.1007/978-3-032-30527-5_30

This paper explores the use of a VM-based approach to managing task execution in embedded multicore systems. Integrating multi-core task management into the VM itself centralises execution control and synchronisation.

Multi-core task management is performed at runtime rather than being handled by mruby/c , a lightweight scripting language runtime for embedded systems. The mrubyc VM handles concurrent task execution across multiple cores.

The key idea behind this work is to assign task ownership on a per-core basis, while also enabling the safe migration of tasks between cores. To achieve this, we introduce a FIFO-based mechanism for passing tasks between cores and transferring TCBs. This is combined with a two-level synchronisation strategy consisting of lightweight, intra-core mutual exclusion and inter-core synchronisation using a global lock. This design allows tasks to be executed in parallel while avoiding excessive memory contention and the need for complex fine-grained locking.

The remainder of this paper is organized as follows. Section 2 reviews related work on embedded multicore systems and virtual machines. Section 3 describes the system model and assumptions. Section 4 presents the design of the proposed VM-based task management approach. Section 5 discusses implementation details. Section 6 evaluates the performance of the system. Finally, Sect. 7 discuss the results and conclude the paper.

2 Related Work

2.1 Power Management in Embedded Multicore Systems

Power management is a widely studied technique for reducing energy consumption in embedded multicore systems with limited power budgets. Many existing approaches focus on hardware-supported mechanisms, such as dynamic voltage and frequency scaling (DVFS), clock gating and core-level power gating. These techniques can effectively lower power consumption during periods of low workload by selectively deactivating idle cores or reducing operating frequency [1,2].

In addition to these hardware-level mechanisms, software-based power management techniques have also been proposed. Operating system-centric approaches, including those implemented in real-time operating systems (RTOS), manage task scheduling and idle states to control power consumption. These approaches typically require the operating system to monitor system load and dynamically decide when to transition cores into low-power states. While effective, such solutions assume the presence of an OS and impose a relatively complex software stack, which may be unsuitable for resource-constrained or bare-metal embedded systems. In bare-metal environments, power management is often implemented directly in C code, with application developers controlling core activation, synchronization and low-power transitions explicitly. While this approach provides fine-grained control, it also tightly couples application logic with hardware-specific power management code, increasing development complexity and the risk of programming errors, particularly in multicore environments with shared memory [3].

2.2 Virtual Machines in Embedded Systems

VMs and scripting language runtimes have been explored as a means of improving productivity and portability in embedded systems. Lightweight VMs such as Lua-based runtimes, MicroPython, and mruby/c are designed to operate under tight memory and performance constraints while providing high-level programming abstractions. These systems execute application code on a VM that manages memory allocation, execution flow, and object lifetime [4–6].

Prior research has demonstrated that VMs can improve software maintainability and development efficiency by abstracting hardware-specific details from application code. In particular, memory allocation and garbage collection in VMs reduce the risk of memory-related errors that are common in low-level programming. However, many existing embedded VMs are designed for single-core execution only and do not address multi-core execution or inter-core task migration.

Some related studies have explored ways of extending VM-supported concurrency through threads or message-passing models. These efforts mainly focus on functional correctness and performance isolation rather than developer performance.

Consequently, little attention has been paid to the interactions between VM-level concurrency and multi-core microcontrollers, or even power management.

2.3 Comparison with Existing Approaches

Compared with existing VMs that support multi-core microcontrollers, the proposed approach is different. Unlike OS-based solutions, it does not rely on a full operating system and can be used in bare-metal environments. Unlike conventional C-based implementations, core control and memory management are not exposed directly to application developers, but are instead abstracted by the VM.

While previous VM-based approaches have simplified application development, they have typically treated power control as an underlying platform feature. This work, however, integrates the power control mechanism directly into the VM design, enabling execution control and memory management to be coordinated within a single abstraction layer.

Furthermore, existing multi-tasking in a multi-core environment often requires explicit synchronisation and careful handling of shared memory at the application level. By managing memory via the VM and restricting unsafe object sharing, the proposed approach absorbs memory contention and synchronisation complexity within the VM. This integration is key to the proposed method compared to previous work, enabling a simpler, safer and more efficient power management model for embedded multicore systems.

3 System Model and Assumptions

This section describes the system model assumed in this work and clarifies the hardware and software assumptions underlying the proposed VM-based power

management approach. The design goals derived from these assumptions are also presented.

3.1 Target System Model

We consider an embedded multicore system consisting of a small number of homogeneous processor cores integrated on a single microcontroller. The target platform is assumed to provide shared on-chip memory accessible by all cores, as well as hardware support for independent core activation and deactivation. Each core can be individually started, stopped, or transitioned into a low-power state by software (Fig. 1).

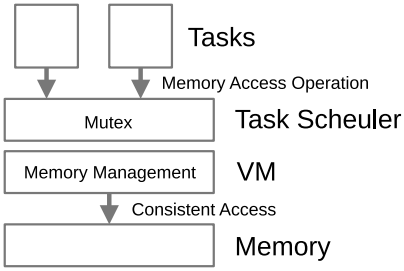

Fig. 1. Single mruby/c Task Model.

The cores execute application code without a memory management unit (MMU), and all memory accesses are performed through a shared address space. As a result, concurrent execution on multiple cores may lead to memory contention and consistency issues if not properly coordinated.

The system is assumed to operate under tight resource constraints in terms of memory size and processing power, which is typical for IoT and deeply embedded applications.

3.2 Software Stack

The software environment assumed in this work consists of a lightweight runtime executing on a bare-metal or minimal system configuration. A scripting language VM runs directly on top of the hardware abstraction layer (HAL) without relying on a full-featured operating system. The HAL provides basic services such as core control, interrupt handling, and access to low-power modes.

The VM executes application logic written in a high-level scripting language and is responsible for managing execution state, memory allocation, and object lifetime. Power management primitives, such as core activation and deactivation, are exposed to applications through VM-provided interfaces rather than direct hardware access.

This design allows hardware-specific details to be encapsulated within the runtime layer.

3.3 Design Goals

Based on the target system model and software assumptions, the following design goals are established:

Safety Ensure memory consistency and correct execution during concurrent operation and power state transitions, without requiring application-level synchronization.

Simplicity Reduce the complexity of application development by abstracting multicore control and power management within the VM.

Efficiency Minimize runtime and memory overhead introduced by the VM-based abstraction to remain suitable for resource-constrained embedded systems.

Generality Design the approach so that it can be applied to different embedded multicore platforms and scripting language VMs with minimal modification.

These goals guide the design of the proposed VM-based power management architecture described in the following section.

4 VM-Based Power Management Design

This section describes the design of the proposed VM-based power management approach for embedded multicore systems. The design integrates task execution control, memory management, and power state transitions within a VM, enabling safe and efficient power-aware operation. Particular attention is given to task management using TCBs and safe task migration across cores [7].

4.1 Design Overview

The proposed approach places the VM at the center of power management in embedded multicore systems. Instead of allowing applications to directly control processor cores and power states, the VM abstracts these operations and exposes them through high-level interfaces. This design enables power management decisions to be tightly coupled with execution flow and memory state, while reducing application-level complexity. By integrating power management into the VM, execution control, memory allocation, and power state transitions can be coordinated at well-defined points. This unified control model is particularly beneficial in multicore environments, where power state transitions must be synchronized with concurrent execution and shared memory access.

4.2 Task Management Using TCBs in mruby/c

To support multiprogramming in resource-constrained embedded systems, mruby/c manages program execution using a TCB-based model(Fig. 2 Left). Each task executed by the VM is represented by a TCB, which encapsulates the execution context of a script program, including the program counter, stack state, and references to task-specific memory.

The VM scheduler operates by selecting a TCB for execution and switching between TCBs at well-defined points, such as instruction boundaries or explicit yielding operations. This cooperative execution model allows mruby/c to precisely control task execution without relying on preemptive scheduling mechanisms, which are often costly or unavailable in small embedded systems. In the proposed design, the TCB abstraction provides a natural unit for coordinating execution and power management. Power-aware decisions, such as suspending execution or stopping a core, are performed only when the execution state of the active TCB is consistent and safely preserved.

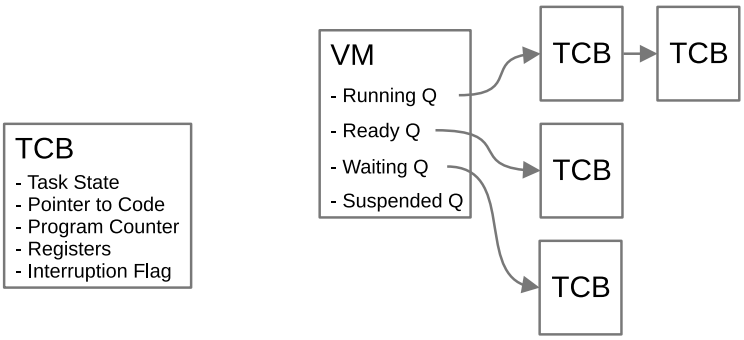

Fig. 2. TCB (Left) and Task State Model (Right).

4.3 Single-Core and Multicore TCB Management

In a single-core system, task management can be handled using a single active TCB, as only one execution context exists at any given time. All tasks are serialized, and memory access contention does not arise. This model simplifies execution control and is sufficient for single-core embedded environments.

Figure 2 Right illustrates the task state model in a single-core system, where only one TCB, which is the first TCB in Running Q, is active and all tasks are executed sequentially.

In contrast, multicore systems require a different strategy. A common approach is to maintain separate TCBs for each core, allowing multiple tasks to execute concurrently. This per-core TCB model improves parallelism and scalability by localizing execution state and minimizing synchronization during normal execution. Using a single shared TCB across multiple cores is impractical, as concurrent accesses to the same execution state lead to frequent memory contention and significantly degrade performance. Therefore, isolating TCBs per core is essential for efficient multicore execution.

4.4 TCB Contention During Task Migration

While per-core TCBs reduce contention during normal execution, additional challenges arise when tasks must be migrated between cores (Fig. 3). Task migration is necessary to support dynamic load balancing and power-aware core control, such as consolidating tasks onto fewer cores before deactivating unused cores. During task migration, ownership of a TCB must be transferred from one core to another. If multiple cores directly access or modify the same TCB during this transition, contention may occur, potentially resulting in inconsistent execution state or excessive synchronization overhead. Therefore, a safe and efficient mechanism for inter-core TCB transfer is required.

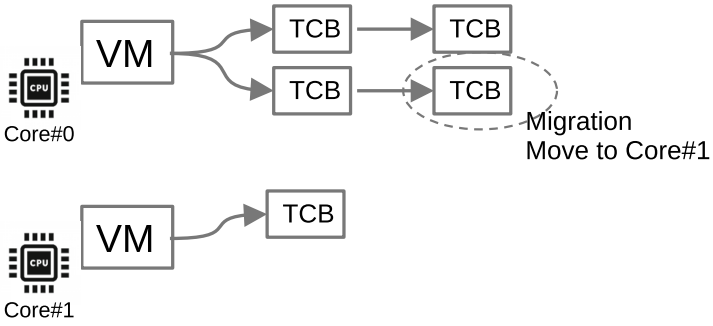

Fig. 3. Task Migration among Cores.

4.5 Task FIFO-Based Inter-core Migration

To address TCB contention during task migration, the proposed design introduces a task FIFO mechanism dedicated to inter-core task transfer. Each core exclusively manages its local TCBs during normal execution. When a task needs to be migrated to another core, the source core enqueues the corresponding TCB into a FIFO structure shared between the cores. The destination core dequeues the TCB from the FIFO and assumes exclusive ownership of the task.

This mechanism ensures that no TCBs are accessible at the same time across multiple cores. To separate task migration from direct TCB sharing, the migrating queue interface can use fine-grained locking or complex synchronisation. This design enables safe task migration while preserving high execution performance (Fig. 4). It also allows task redistribution and power management decisions to be implemented within the VM, ensuring that cores can be activated or deactivated dynamically without application developers being exposed to low-level concurrency concerns.

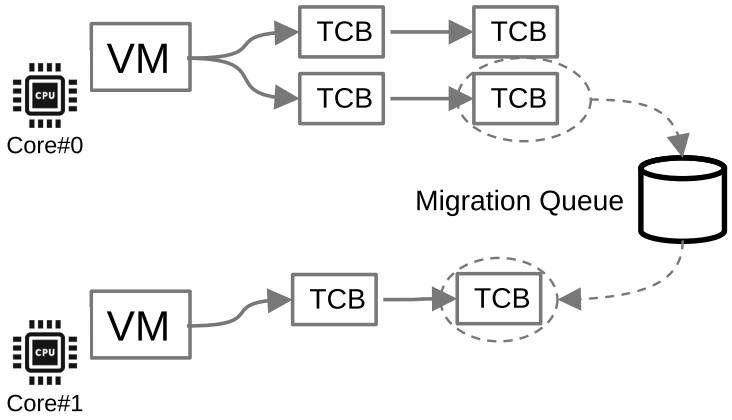

Fig. 4. Task Migration Queue.

5 Implementation

This section describes the implementation of the proposed VM-based multitasking system on a multicore microprocessor. The prototype system uses mruby /c and is implemented on a dual-core embedded microcontroller (RP2350) in a bare-metal environment. Our focus is on task management based on TCBs, inter-core task migration using a task migration queue and the synchronization mechanisms required to ensure the system operates correctly.

5.1 Prototype System Overview

The prototype system is designed for a dual-core embedded microcontroller and runs on bare metal rather than an operating system. One core is primarily responsible for system control, including VM initialisation, interruptions and power management coordination, while the other core executes tasks assigned by the VM.

The mruby/c VM runs on both cores and each core independently maintains its own execution context and local task set. A lightweight hardware abstraction layer (HAL) provides access to the low-level hardware functions, such as core activation, deactivation, mutual exclusion and power state control.

5.2 TCB Management and Scheduling

In the proposed implementation, each script task executed by the VM is represented by a TCB. A TCB data structure contains a task's execution state, including the program counter, stack information and references to VM-managed memory. During normal execution, each core maintains a local list of TCBs that it exclusively owns, in a manner similar to a single-core environment.

Context switches only occur at VM-defined safe points, ensuring that the state of the TCBs remains consistent and avoiding unexpected preemption. By restricting direct access to TCBs to the owning core, the system minimises memory contention during multicore execution and avoids the need for fine-grained locking within the VM.

5.3 Task FIFO for Inter-core Task Migration

To support dynamic task redistribution across cores, a task FIFO is introduced for inter-core task migration. The task FIFO is implemented as a bounded queue shared between cores and stores references to TCBs that are pending migration. When the VM determines that a task should be moved to another core—for example, to consolidate tasks before deactivating a core—the source core enqueues the corresponding TCB into the task FIFO and releases ownership. The destination core periodically checks the FIFO and dequeues TCBs when available, after which it assumes exclusive ownership of the tasks. This FIFO-based approach ensures that a TCB is accessed by at most one core at any time, preventing concurrent modification and eliminating direct TCB sharing between cores. As a result, task migration is performed safely without introducing complex synchronization mechanisms.

5.4 Integration with Power Management

Power management operations are integrated into the VM's task management flow. Before a core is transitioned into a low-power state, the VM migrates all remaining tasks on that core to another core using the task FIFO. Once the local TCB list becomes empty, the VM invokes the HAL to deactivate or suspend the core.

Similarly, when workload increases, a previously deactivated core can be reactivated and assigned tasks via the same FIFO mechanism. By performing power state transitions only at VM-defined safe points, the system ensures that no task execution state is lost and that memory consistency is preserved.

5.5 Mutual Exclusion and Synchronization

To ensure correct behavior in a multicore environment, the implementation employs two levels of mutual exclusion mechanisms: intra-core mutual exclusion and inter-core mutual exclusion using a giant lock.

Intra-core Mutual Exclusion. Within a single core, the VM uses lightweight mutual exclusion when manipulating internal data structures such as local TCB lists and scheduler state. Because mruby/c relies on cooperative multitasking, concurrent execution within a single core does not occur. Therefore, intra-core mutual exclusion is only required to protect critical sections during task scheduling or TCB state updates.

This intra-core synchronization is implemented using short critical sections, such as temporarily disabling interrupts. The limited scope of these critical sections ensures low overhead and deterministic behavior.

Inter-core Mutual Exclusion Using a Giant Lock. In contrast, inter-core mutual exclusion is required for operations that involve shared resources across cores. The implementation employs a giant lock shared by all cores to serialize access to the following resources:

- Hardware registers related to power control and core management
- Global variables shared between cores
- Shared data structures such as the task FIFO

Before accessing any of these shared resources, a core must acquire the giant lock. This approach ensures that hardware state and global data remain consistent and prevents race conditions between cores.

Although the giant lock introduces serialization, it is used only for infrequent operations such as task migration and power state transitions. Normal task execution and local TCB management are performed without acquiring the giant lock, keeping contention low.

Coordinated Use of Both Lock Levels. When the VM performs operations that involve both task management and power control, the two levels of mutual exclusion are used in a coordinated manner. First, intra-core mutual exclusion is applied to ensure the consistency of local TCB state. Then, the inter-core giant lock is acquired before accessing shared resources or hardware registers.

This hierarchical locking strategy ensures that:

- TCBs are never concurrently modified by multiple cores
- Power state transitions are performed atomically
- Global system state remains consistent across cores

As demonstrated in the evaluation section, the overhead introduced by this synchronization strategy remains acceptable for resource-constrained embedded systems.

6 Evaluation

This section evaluates the proposed VM-based multicore execution and task management mechanisms on a real embedded platform. The evaluation focuses on execution performance and synchronization overhead to assess the practicality of the proposed design.

6.1 Experimental Setup

All experiments were conducted on a dual-core embedded microcontroller operating at a fixed clock frequency. The system was configured in a bare-metal environment without a full-featured operating system. The proposed implementation using mruby/c was compared with a conventional C-based implementation that executes tasks directly on multiple cores without a VM. Execution time was measured using on-chip timers. Each experiment was repeated multiple times, and average values were reported to reduce measurement noise.

6.2 Workloads

To evaluate the system under representative operating conditions, the following workloads were prepared:

Single-core workload: All tasks are executed on a single core while the other core remains inactive. Parallel workload: Tasks are distributed across both cores to evaluate parallel execution performance. Dynamic workload: Tasks are dynamically redistributed between cores, triggering task migration via the task FIFO.

These workloads represent typical embedded use cases in which task parallelism and runtime task redistribution are required.

6.3 Performance Evaluation

Execution performance was evaluated by measuring task completion time under each workload. Figure 5 shows the relationship between the number of tasks and execution time for both single-core and dual-core configurations. Every task is categorised as a heavy load. In this evaluation, each task involves calculating a Fibonacci number using a recursive call. Compared with single-core execution, parallel execution using two cores achieved a performance improvement of up to 180%, single core execution time was 5.8×10^8 [μs], while dual-core execution reduced it to 3.2×10^8 [μs].

The VM-based implementation introduced a moderate runtime overhead relative to the C-based implementation due to instruction interpretation and VM-level task management. However, this overhead remained within 1%, indicating that the VM abstraction is feasible even in resource-constrained embedded environments.

The performance impact of task migration was also evaluated. Migrating a task between cores via the task FIFO incurred an overhead of 1.1–20.0 [ms] per migration. It was observed that the migration overhead varies depending on the runtime load of the destination core. When the destination core operates under low or moderate load, the migrated task is quickly incorporated into the local scheduling queue, and the associated TCB is updated with minimal delay.

In contrast, when the destination core is under heavy load, the migration overhead increases. This behavior is caused by the VM's cooperative execution model: when the VM load is high, the entire allocated time slice is consumed

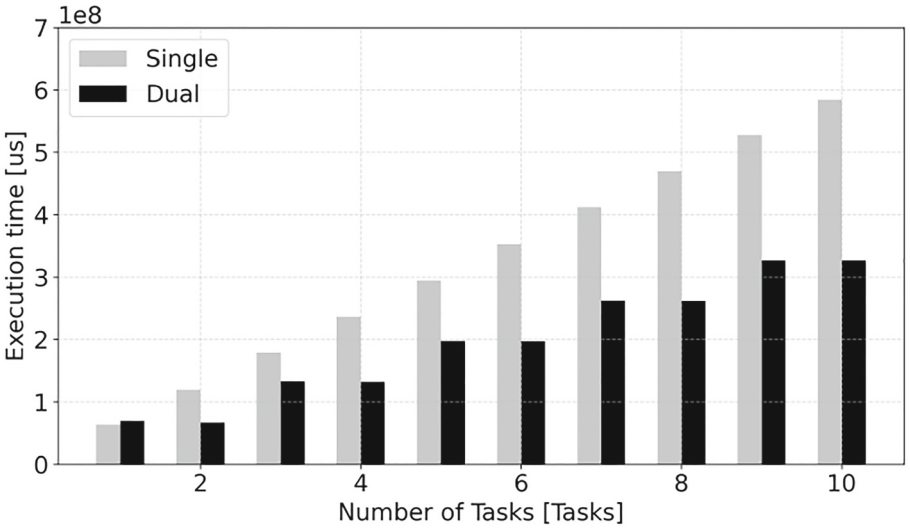

Fig. 5. Execution Performance Evaluation.

by task execution, and TCB update operations related to task migration are deferred until a scheduling safe point is reached. As a result, the completion of TCB state updates required for task migration may be delayed, leading to increased migration latency under high VM load.

Despite this variability, the observed migration overhead remains small relative to typical task execution times. Therefore, dynamic task redistribution using FIFO-based migration does not significantly degrade overall system performance in practical embedded workloads.

6.4 Synchronization Overhead

Synchronization overhead was analyzed by measuring the frequency and duration of mutual exclusion operations. The proposed implementation employs two levels of synchronization: intra-core mutual exclusion and inter-core mutual exclusion using a giant lock. Intra-core mutual exclusion incurs minimal overhead because task execution within a core is cooperative and serialized. Inter-core synchronization using the giant lock occurs primarily during task migration and hardware access. Measurements show that giant lock acquisition events are rare, and that the time spent holding the lock accounts for only a small proportion (a maximum of 20 ms) of the execution time. The holding time depends on the load on each core; especially when executing a heavy load, the holding time increases. However, since the giant lock is only held during infrequent operations such as task migration and power state transitions, the overall impact on performance is minimal. These results demonstrate that the use of a simple giant lock is sufficient for ensuring correctness while keeping synchronization overhead low.

6.5 Summary of Evaluation Results

The evaluation results show that the proposed VM-based design enables efficient multicore execution with acceptable overhead. Task management using per-core TCBs and FIFO-based inter-core migration provides good parallel performance while avoiding excessive synchronization costs. Overall, the proposed implementation demonstrates that VM-level task management and synchronization can be practically applied to embedded multicore systems without sacrificing performance.

6.6 Discussion

The proposed VM-based approach demonstrates that multicore task management can be effectively centralized within a lightweight virtual machine in embedded systems. By leveraging the TCB-based execution model of mruby /c , the VM provides a clear abstraction for task execution, synchronization, and core coordination, significantly reducing application-level complexity. A key advantage of the design is the combination of per-core TCB ownership and FIFO-based inter-core task migration, which enables parallel execution while avoiding frequent memory contention. In addition, the two-level synchronization strategy—lightweight intra-core mutual exclusion and inter-core synchronization using a giant lock—offers a practical balance between correctness and implementation simplicity. The evaluation results indicate that the resulting synchronization overhead remains low in a dual-core environment. Nonetheless, several limitations remain. The use of a giant lock may limit scalability as the number of cores increases, and VM-based execution inevitably introduces runtime overhead compared with native implementations. Furthermore, this study focuses primarily on execution performance and synchronization behavior; quantitative power consumption evaluation is left for future work. Despite these limitations, the proposed design principles are not specific to mruby/c or a particular hardware platform. The presented approach provides a general framework for VM-based multicore task management that can be adapted to other lightweight runtimes and embedded multicore systems.

7 Conclusion

This paper presents a VM-based approach to task management and synchronization for embedded multicore systems. By integrating multicore execution control into the virtual machine, the proposed design abstracts complex concurrency and synchronization issues away from application developers. The proposed method leverages the TCB-based execution model of mruby/c , introduces per-core task ownership, and employs a FIFO-based mechanism for safe inter-core task migration. A two-level mutual exclusion strategy combining intra-core synchronization and an inter-core giant lock ensures correctness while keeping overhead low. Experimental evaluation on a real dual-core embedded platform

demonstrated that the proposed approach achieves efficient multicore execution with acceptable performance and synchronization overhead. These results indicate that VM-based task management is a practical and effective solution for embedded multicore systems. Future work includes extending the approach to systems with higher core counts, conducting detailed power consumption evaluations, and exploring integration with real-time operating systems and adaptive scheduling policies.

References

1. Qualcomm Technologies, Inc.: Dynamic voltage and frequency scaling. Qualcomm Documentation. https://docs.qualcomm.com/bundle/publicresource/topics/80-70018-30/dcvs.html, updated 10 Apr 2025, Accessed 30 Mar 2026
2. Massari, G., Terraneo, F., Zanella, M., Zoni, D.: Towards Fine-Grained DVFS in Embedded Multi-core CPUs. In: Berekovic, M., Buchty, R., Hamann, H., Koch, D., Pionteck, T. (eds.) ARCS 2018. LNCS, vol. 10793, pp. 239–251. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77610-1_18
3. Kim, S.H., et al.: C-LockâĂŕ: energy efficient synchronization for embedded multicore systems. IEEE Trans. Comput. **63**(8), 1962–1974 (2014). https://doi.org/10.1109/TC.2013.84
4. Silangern, H., Tanaka, K.: Internet of things implemented with Mruby. In: Gervasi, O., Murgante, B., Garau, C., Taniar, D.C., Rocha, A.M.A., Faginas Lago, M.N. (eds.) Computational Science and Its Applications(ICCSA) 2024 Workshops. ICCSA 2024. Lecture Notes in Computer Science, vol. 14819. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65282-0_11
5. Tanaka, K., Higashi, H.: Mruby—rapid IoT software development language. In: 2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA), Siem Reap, Cambodia, pp. 415–418 (2017). https://doi.org/10.1109/ICIEA.2017.8282880
6. Go, S., Takuo, W., Sosuke, M.: Mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices. In: MPLR 24: 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, pp. 41–47 (2024). https://doi.org/10.1145/3679007.3685064.
7. Yao, J., Mo, L., Han, J., Niu, D.: Task scheduling and temperature optimization codesign for multi-core embedded systems. In: International Symposium of Electronics Design Automation (ISEDA), vol. 2025, pp. 683–688. Hong Kong, China (2025). https://doi.org/10.1109/ISEDA65950.2025.11100782