A Global Dynamic Load Balancing Mechanism with Low Latency for Micokernel Operating System

Qinyun Tan
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
qinyun_tan@163.com

Kun Xiao
School of Information and Software Engineering
University of Electronic Science and Technology of China
Chengdu, China
xiaokun@uestc.edu.cn

Wen He
Chongqing Changan Automobile Corporation
Chongqing, China
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
hewen@changan.com.cn

Pinyuan Lei
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
leipinyuan1996@126.com

Lirong Chen
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
lrchen@uestc.edu.cn

Abstract—As Internet of Things(IOT) devices become intelli-gent, more powerful computing capability is required. Multi-core processors are widely used in IoT devices because they provide more powerful computing capability while ensuring low power consumption. Therefore, it requires the operating system on IoT devices to support and optimize the scheduling algorithm for multi-core processors. Nowadays, microkernel-based operating systems, such as QNX Neutrino RTOS and HUAWEI Harmony OS, are widely used in IoT devices because of their real-time and security feature. However, research on multi-core scheduling for microkernel operating systems is relatively limited, especially for load balancing mechanisms. Related research is still mainly focused on the traditional monolithic operating systems, such as Linux. Therefore, this paper proposes a low-latency, high-performance, and high real-time centralized global dynamic multi-core load balancing method for the microkernel operating system. It has been implemented and tested on our own microker-nel operating system named Mginkgo. The test results show that when there is load imbalance in the system, load balancing can be performed automatically so that all processors in the system can try to achieve the maximum throughput and resource utilization. And the latency brought by load balancing to the system is very low, about 4882 cycles (about 6.164us) triggered by new task creation and about 6596 cycles (about 8.328us) triggered by timing. In addition, we also tested the improvement of system throughput and CPU utilization. The results show that load balancing can improve the CPU utilization by 20% under the preset case, while the CPU utilization occupied by load balancing is negligibly low, about 0.0082%.

Keywords: microkernel, load balance, low-latency, multi-core, high-performance

This paper is funded by the project of the state grid corporation of china in 2020 " Integration Technology Research and Prototype Development for High End Controller Chip"(5700-202041264A-0-0-00).

I. Introduction

A. Background

Recently, with the development of the chip manufacturing industry, single-core processors have been unable to meet the requirements of mainstream embedded systems for high performance, low power consumption, and cost-effectiveness. Despite Moore's law, processor performance increase as the number of transistors per unit area of the chip increases [1], but this approach generates too much heat. In addition, relevant research shows more than twice the manufacturing cost for only 20% overall performance improvement [2]. Therefore, at the beginning of the 21st century, chipmakers integrated two or even more processor cores on a single chip. The parallel work of the cores improved the chip's overall performance, which makes the high-performance chips more cost-effective. Multi-core processors became the primary trend in the development of processors. Meanwhile, the operating systems for multi-core were born and became the research hotspot in the industry [3].

In the operating system field, the global market is occupied by Windows, Linux, Android, MacOS, and IOS. With the development of the embedded field, embedded devices require more and more security, scalability, and lightness of the kernel. With their large code size, low security, and low scalability, traditional operating systems do not meet the requirements of embedded devices. At this time, microkernel operating systems reappeared in the vision of researchers. The main idea of microkernel operating systems is to reduce the number of functional modules implemented by the kernel so that the kernel provides only the most basic kernel services, such as task scheduling, IPC(Inter-Process Communication), and I/O control, while system services such as the file system, network protocol stack, and hardware drivers are implemented at the user level. Recently, Huawei has proposed HarmonyOS, a distributed operating system based on microkernel for complete scenarios, and launched several embedded devices with HarmonyOS. Based on the above features of the microkernel and the triumph achieved by Huawei, it is widely believed in the industry that the simpler, more secure, and more reliable microkernel operating system will become the mainstream trend in the development of embedded devices compared with traditional operating systems [4].

As mentioned earlier, multi-core processors will become the major trend in processor development. However, most microkernel operating systems are not mature enough to support multi-core processor platforms and are hard to adapt to the trend of multi-core embedded devices. Load balancing is vital as an indispensable part of single-core to multi-core to maximize the performance of multi-core embedded devices. However, scheduling tasks between the cores of a multicore processor in real time is an NP-hard problem [5]. Thus, load balancing is the main concern in any real-time multi-core system.

The current research on load balancing of operating systems mainly focuses on traditional monolithic kernel operating systems, while the research on load balancing for microkernel operating systems is lacking. Therefore, conducting in-depth research on microkernel operating systems and designing reliable and efficient load balancing methods is on our way.

B. Multi-core processors

Single-core processors have encountered bottlenecks in both performance and power consumption in the development process. At the beginning of the 21st century, chip manufacturers proposed a variety of improvement strategies. One way is to integrate multiple single-core processors with similar functions on one chip, called a multi-processor chip. Another way is to integrate multiple CPU cores on a single processor, called chip multi-processor(CMP) [6] .

For multi-core processor chips, they can be divided into Homogeneous Multi-core Processor and Heterogeneous Multi-core Processor according to the consistency of the structure of each processor core integrated with them [7].

In the homogeneous multi-core processor, each processor core has the same structure, function, and level. In that case, each core accomplishes the tasks assigned over by the system as a computing resource. Depending on the performance of individual cores, this class of processors can be further divided into high-performance processors and low-power processors [8]. For example, the Core i9-10900K processor developed by Intel is a typical high-performance processor, which integrates ten identical high-performance cores to meet the performance requirements of a single core in the desktop computer server environment. In addition, the Cortex-A9 MPCore chip [9] provided by ARM is a typical low-power processor, which integrates four A9 processors to meet the demand of low-power, low-cost and real-time application.

Multiple cores of different types are integrated simultaneously in heterogeneous multi-core processors, usually containing one or more processor cores for particular industries. For example, the Qualcomm Snapdragon 888 processor uses a combination of large and small cores of ARM's latest super large cores, Cortex X1, A78, and A55. Since the study in this paper is based on homogeneous multi-core processors, we will not elaborate too much here.

C. Microkernel Architecture

There are many different ways to classify modern operating systems. Nevertheless, the most commonly accepted kernel architecture implementations are classified into three types: Monolithic kernel, Microkernel, and Hybrid kernel. The difference is shown in the Fig. 1:

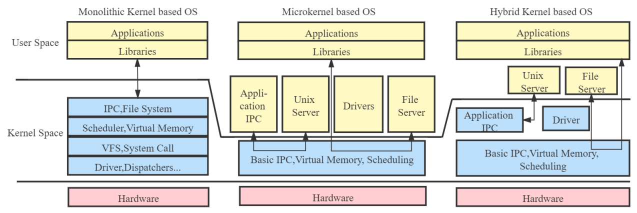

Fig. 1. Comparison of monolithic/micro/hybrid kernel operating systems.

The mainstream operating systems such as Windows, Linux, Android, and IOS all use Monolithic kernel architecture. The Monolithic kernel adopts a layered design method. All essential system services such as thread management, memory management, interrupt management, I/O communication, and file system are run in the kernel space. All modules in the kernel can directly use the essential services provided by the others through function calls, which brings high performance but also leads to a massive amount of kernel code that is expensive and difficult to maintain.

To overcome a series of shortcomings of the Monolithic kernel, the microkernel architecture was proposed in the 1980s. The main idea of the microkernel is to simplify the kernel functions, only provide the most basic kernel services such as task management, inter-task communication, and terminal management, while the file system, network protocol stack, and device driver are all implemented in the userspace. Therefore, compared with the monolithic kernel, the microkernel code size is smaller, facilitating the code's maintenance and improving the system's stability. In addition, the new system services will run in user space, so they can be extended as if they were a standard application. This modular development makes the kernel has good scalability. Finally, some complex and code-heavy system services such as networking and hardware drivers run in a different address space from the kernel. This makes it easier to isolate the kernel and improve the system's overall security so that errors in one module do not affect other modules in the kernel.

In summary, the microkernel has the advantages of small code size, high security, and high scalability. The Mginkgo used in this article is a microkernel designed and implemented by our lab based on the third-generation microkernel seL4.

D. Load Balancing

Compared with single-core processors, multi-core processors can provide better performance and efficiency. Therefore, more and more multi-core processors are used in embedded systems. Load balancing is a critical issue to maximize the performance of multi-core processors.

Load balancing [10] means that all processors in an embedded multi-core system can achieve the maximum throughput and resource utilization as much as possible, which requires that each processor can perform tasks in a balanced manner with the same load. For the multi-core embedded system, it is important to ensure load balancing of CPU cores while also taking the real-time requirements into account.

In recent years, the scheduling algorithms proposed for load balancing of multi-core processors are mainly optimized for performance [11]–[13], energy efficiency [14], [15], and fairness [16]. At present, load balancing algorithms have the following mainstream classifications [17]:

1) **Partition and Global:** Partition load balancing ensures that multiple threads use CPU resources reasonably by assigning appropriate time-slice sizes to different threads. In the partition load balancing method [5], tasks are assigned to the core statically, and migration between cores is not allowed. The main advantage of this approach is that there is no migration overhead. However, it has two significant disadvantages: Firstly, such schemes are inflexible and cannot easily accom-modate dynamic tasks without a complete re-partition. Secondly, the optimal allocation of core tasks is an NP-hard problem [18]. Global load balancing is specifically for multi-core processors. It needs to set reasonable time-slices for different threads on the same core and maintain a relatively balanced task load on different processors. In the global scheduling strategy, tasks are allowed to migrate between cores as needed. In recent years, several optimal global scheduling strategies have been proposed [19]–[21]. Although these schemes strive to overcome the limitations of local scheduling, they add migration overhead to the task. In the context of real-time systems, increasing overhead changes the timing behavior of tasks, thereby affecting the timing predictability of the system.
2) **Static and dynamic load balancing:** In static load balancing, tasks are assigned to designated cores according to system conditions when the code is compiled. In dynamic load balancing, tasks are redistributed through inter-core migration according to system conditions and balancing strategy during the task run-time [22]. Researchs have shown that dynamic load balancing has 30%-40% performance improvement compared with static load balancing [23].
3) **Centralized and distributed:** In centralized load balancing, one core runs the load balancing process to complete all steps of system load balancing. In distributed load balancing, each core can decide whether to run the load balancing program according to the situation and make load balancing decisions and initiate task migration on its own. Dynamic global load balancing is generally divided into three steps: load monitoring, balancing decision, and task migration. Load monitoring monitors the task load on different cores in the system and determines whether there is a load imbalance. The balancing decision determines whether load balancing is needed through task migration between cores based on the real-time load situation obtained from load monitoring. For task migration, the following parameters are generally required to be decided:
   - When should a task be migrated?
   - Which task should be migrated?
   - Where should the task be migrated to?

E. CONTRIBUTION

The main contribution of this paper is to design a multi-core load balancing approach for microkernels and to implement and test it in our own microkernel operating system named Mginkgo. Compared with traditional load balancing methods, this method is based on the characteristics of the Mginkgo microkernel and uses a centralized, dynamic, and global management approach with low latency, high performance, and high real-time characteristics.

The structure of this paper is organized as follows: Section II introduces the related work about microkernel and load balancing. Section III describes our design and implementation of the load balancing module. Section IV presents a series of tests after we implemented load balancing on the Mgkingo. Section V is the conclusion of our work.

II. Related Work

A. Mginkgo microkernel

The Mginkgo microkernel is designed and implemented by our lab based on the third-generation microkernel seL4. The purpose is to meet the industry's increasing requirements for industrial control-oriented operating systems regarding security, real-time, and reliability. In this microkernel, the main functional modules mainly include five modules: capability subsystem, task management, memory management, IPC module, and interrupt management. The functions of each module are as follows:

1) **Capability subsystem:** It is used to record the relevant permissions that each task has to access kernel resources and complete the task's permission check when it initiates an access request.
2) **Task management:** It can also be called thread management in the microkernel, which is mainly used for the organization and scheduling of all tasks in the system.
3) **Memory management:** On the one hand, it manages memory resources, and on the other hand, it realizes the conversion between physical addresses and virtual addresses.

4) **IPC module:** The inter-task communication module mainly implements two basic ways of communication between tasks: time notification mechanism and message transfer mechanism.
5) **Interrupt management:** The interrupt management module mainly manages to interrupt resources, with interrupt registration, notification, response, enable and disable functions.

Besides, it is worth mentioning that there is a thread called rootserver in the Mginko. It is a thread created by the kernel during the start-up phase of the Mginko, and the rootserver manages the system resources. In task scheduling, the rootserver also handles the creation of threads, and the kernel is only used for detection.

The Mginkgo implements the five modules mentioned above and can run stably on single-core processor chips, but the multi-core support is insufficient. Load balancing is vital as an integral part of single-core to multi-core. In this paper, we design and implement a load balancing module based on the Mginkgo.

B. Load Balancing

At present, the main goal of multi-core processors is to distribute the workload among each processor's core so that each core's utilization rate is equal or approximately equal, ensuring that system resources are fully utilized, obtaining the maximum system throughput, and shrinking the response time of the task. Many researchers have proposed different strategies to schedule the tasks and balance the workload of the system and have applied them to different areas, such as parallel computing [24], fog computing [25], cloud computing [26], and real-time environments [27].

Load balancing methods can be classified a s s tatic load balancing and dynamic load balancing. For static load balancing, the computer must distribute tasks among multiple cores using a priori task information before execution, and the load distribution remains constant at run-time. HASS [28] is a static load balancing decision algorithm. It analyzes the program in advance before load balancing with the help of feedback optimization techniques of the compiler. The analysis mainly includes information about the program's accesses on different configuration cores as a basis for whether the program benefits from a large core or not.

For dynamic load balancing, instead of using a priori task information, the computer makes dynamic task assignments by collecting relevant task information during run-time.

In mainstream Linux operating systems, load balancing in the scheduler is designed for symmetrical multi-processing(SMP) to achieve a balanced state of processing among the cores by distributing the load evenly among them. Traditional load balancing is implemented using both pull and push. Take the pull method as an example. When the current CPU run queue is empty, load balancing is triggered, and the scheduler finds t he C PU w ith t he h eaviest l oad a nd moves the tasks in the CPU run queue to the idle core. In Linux, a dynamic load balancing algorithm called Completely Fair Scheduler(CFS) is used. The basic idea of this algorithm is to simulate an ideal multitasking processor on real hardware so that all tasks get the CPU as somewhat as possible.

In Android OS, Brain Fuck Scheduler(BFS) is used as the task scheduling algorithm. BFS assigns a time-slice and a virtual deadline to each task, and the scheduler picks the task with the smallest virtual deadline to run each time.

Kang and Waddington [29] proposed a Load Balancing Task Partitioning(LBTP) algorithm, which aims to distribute the computed load so that every core has the same workload. Their idea is to use a task partitioning mechanism that leads to good schedule ability. And then apply other partitioning steps to improve load balancing testing while ensuring that the solution meets the deadline. The algorithm works in three steps by considering independent periodic tasks. The first step is to sort the tasks in descending order of task utilization. After that, the task set is divided according to the first available core to which they can fit so that the utilization of each core is less than or equal to the first one . At last, the tasks in the kernel are re-allocated to reduce the imbalance between the cores.

Wang et al. [30], and Rehman et al. [13] proposed a centralized dynamic scheduling algorithm for multi-core processors, which dynamically adjusts processor parameters according to the run-time state of the application. Based on CFS, Wang et al. proposed a new Heterogeneity-aware Fair Scheduler(HFS) algorithm to add a centralized task queue to support rapid allocation and adjustment of logical cores. Rehman et al. proposed a scheduling technique based on a stable matching algorithm to maintain a priority table of dynamic threaded tasks and cores, which is used for scheduling. Liu et al. [31] proposed an iterative scheduling algorithm to improve throughput while meeting power-consuming threads.

Becchi et al. [32] and Craeynest et al. [12] proposed a dynamic scheduling algorithms to optimize the scheduling strategy according to the run-time state of the application. Becchi et al. allocates threads to different types of cores to run for a while and periodically performs IPC sampling. Thread migration is performed according to the IPC speedup ratio running on the big and small core. Craeynest et al. establishes a stack model based on periodic statistical performance events (including the number of LLC missing, the number of instructions, the distribution of dependent distances between instructions, etc.), and combines the hardware architecture parameters to make Memory level parallelism(MLP) core Instruction Level Parallelism(ILP) predictions, and make scheduling decisions based on the prediction results. Li et al. [33] proposed a heterogeneous-aware load balancing strategy to ensure that the load is proportional to the power consumption. The system execution performance is improved by a strategy that prioritizes large cores, predicts the cost of thread migration by monitoring the resident working set of threads online, and optimizes thread migration based on the migration cost. Jadon et al. [34] proposed a harmonic-aware load balancing(HALB) algorithm which aims to balance the workload amongst the cores by grouping harmonic tasks together and assigning these groups to different cores under deadline monotonic scheduling policy. The algorithm uses the concept that grouping the harmonic tasks together means achieving utilization closer to one, which overall improves the efficiency of the cores and performance of the multi-core system.

Different types of load balancing algorithms have their advantages and disadvantages. Static load balancing algorithms and partitioned load balancing algorithms have no task migration overhead. However, they cannot easily accommodate dynamic tasks, and optimal task assignment is an NP-hard problem. Dynamic and global load balancing algorithms can easily contain dynamic tasks and load balancing more effectively, but bringing the time consumption of task migration. The advantage of centralized load balancing algorithms is easy to control and simple to implement. However, the disadvantage is also evident that the load balancing itself can be a performance bottleneck. Distributed load balancing algorithms have the advantage of implementing some more sophisticated and complex algorithms but bring more overhead because a load balancer is needed on each core.

From the perspective of fully utilizing the performance of multiple cores, load balancing of multi-core processors is essential to fully utilize each core so that the system's performance can be improved with good response time. The current mainstream load balancing algorithms are monolithic kernel-oriented. Due to the architectural differences between microkernels and monolithic, the load balancing algorithm for microkernels needs to be designed and implemented according to it's characteristics. In this paper, we design a centralized global dynamic load balancing method based on Mginkgo.

III. OUR WORK

As mentioned earlier, our goal is to design a load balancing module and implement it on the Mginkgo. In Mginkgo, since the rootserver manages all the kernel resources, task creation, and resource allocation, it is more suitable to adopt a centralized processing method. The core with the rootserver performs the global load balancing. Therefore, we adopt a centralized global dynamic load balancing strategy and divide it into three parts: load balancing, balancing decision, and task migration. The process of load balancing is shown in Fig. 2.

A. Load Monitoring

The load monitoring section will collect the task load on different cores in the system and determine whether a load imbalance has occurred. In the kernel, we use the load_state_t structure to record the load information. As shown in Fig. 3, this structure records the number of ready tasks on the core, the number of sleep tasks, the currently running tasks, the queue where the sleep tasks are located, the sum of the time slice sizes of all ready tasks, and the size of each ready task and its time slice on the core.

The process of load monitoring is the process of populating the load_state_t structure based on the real-time load of the core. The load monitoring algorithm applied in this paper uses

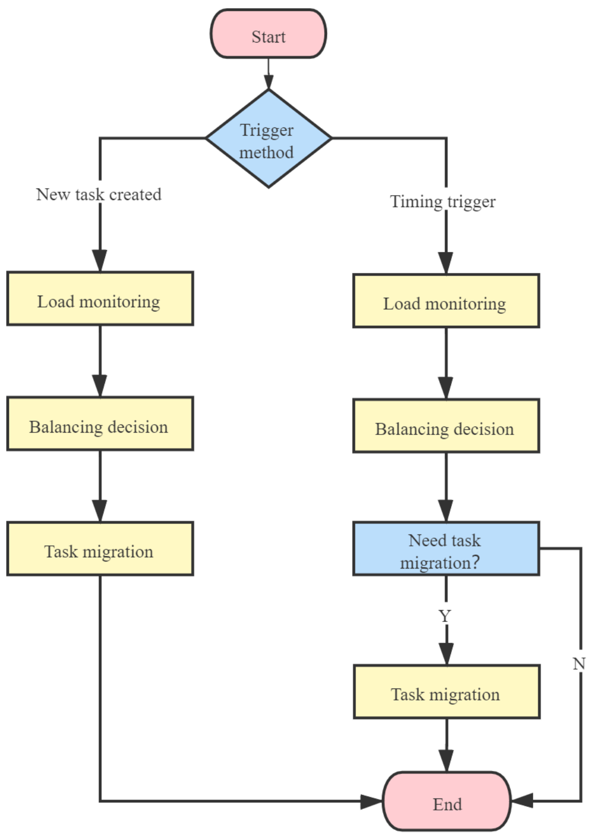

Fig. 2. Load Balancing Flowchart.

```
typedef struct _load_state_t {
    unsigned int task_ready_count;
    unsigned int task_sleep_count;
    tcb_t* pcurrent;
    tcb_t* ppend;
    unsigned int totaltime;
    task_info_t task_info[CONFIG_MAX_TASKS];
    }load_state_t;
```

Fig. 3. The definition of the load_state_t.

the time-slice size of the ready task on the core as the load evaluation factor and is defined as follows.

Definition 1: The sum of the time-slice sizes of all ready tasks on a single processor is regarded as the processor load which is denoted by L. And T denotes the load proportion threshold. In our design, the value of T is 20%. A denotes the average processor load of all CPUs in the current system (except where the rootserver is located). The value of L/A is referred to as the degree of load on the core. Obviously, the larger the value, the heavier the load on the core. The load state S of the core can be determined by equation 1.

$$S(L) = \begin{cases} heavy & L > A*(1+T) \\ normal & A*(1+T) \geq L \geq A*(1-T) \\ light & L < A*(1-T) \end{cases} \quad (1)$$

In equation 1, heavy means the current processor is in a heavy load state, and the tasks need to be migrated out. Normal means the current processor is in a normal load state, and no need for load adjustment. Light means the current processor is in a light load state, and there is a waste of CPU resources, and the load can be increased by adding new tasks.

Load monitoring can be triggered in two ways. First, it is triggered when a new task is created so that the relatively free core can be found to allocate that task. Second, it is performed periodically in the rootserver to get the load status of other cores and determine whether load balancing is needed through inter-core task migration.

B. Balancing decision

From the above load monitoring, it can be found that there are two triggers for the balancing decision, triggered when a new task is created and triggered at timing. Therefore, the balancing decision algorithm needs to be designed for these two cases separately.

When a new task is created, the rootserver will use the analysis results of load monitoring to get the core ID that is relatively idle in the system and then assign the task to it. When triggered at timings, the rootserver analyzes the load on each core in the system and tries to get the task that will be migrated. The algorithm flow is shown in Fig. 4.

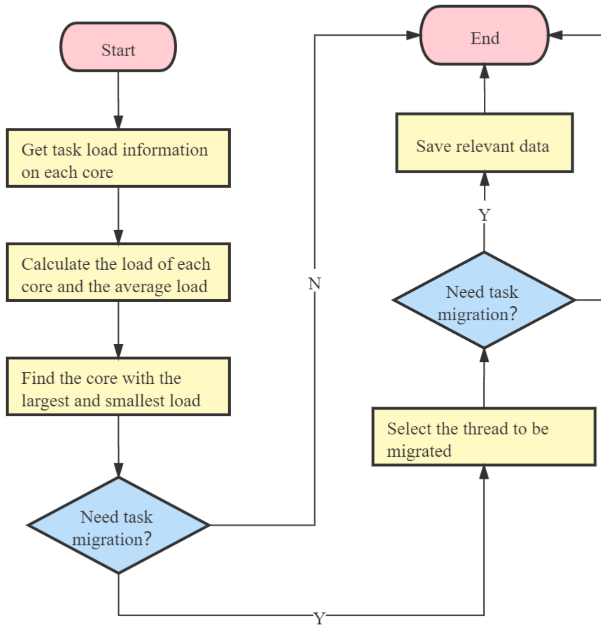
Start
Get task load information on each core
Calculate the load of each core and the average load
Find the core with the largest and smallest load
Need task migration?
N
Y
Select the thread to be migrated
Need task migration?
Y
N
Save relevant data
End

Fig. 4. Balancing decision algorithm.

The steps of the algorithm are shown as follows.

1) The rootserver traverses the load of all cores, gets all ready tasks running on them and their time-slice sizes, and saves them in a two-dimensional array task_info_cpus with the task_info_t structure as a member of the array.
2) Based on task_info_cpus, the sum of the time-slice sizes of the ready tasks on each core is calculated and saved in the integer array named sum. Also, the average load of the cores is calculated and saved in the variable named ave.
3) Find out the most heavily loaded core and the least loaded core in the sum array and save their IDs in the variables named max and min.
4) Determine if inter-core migration of tasks is needed. If the state of the core with the heaviest load in the system is not heavy or the state of the core with the lightest load is not light, which means no load balancing is needed, there is no need to go to step (5) and the function will return null directly. Otherwise, go to step (5).
5) The thread task with the smallest time-slice size is selected from the most heavily loaded core, and it is determined whether load balancing can be achieved by migration of this task according to the following method.
    - If the load level of the currently most heavily loaded core is greater than the load level of the currently least heavily loaded core after migrating to the task thread, it means that the load level of the most heavily loaded core in the system can be reduced after completing this task migration.
    - Otherwise, it means that migrating the task thread only eliminates the overload on one core but adds a new overloaded core, which is insufficient to alleviate the global load imbalance. This step will directly return to null.
6) Write the max and min values to the memory area pointed by the from pointer and to pointer. Furthermore, return the task thread found in step (5) as the function return value.

These are the steps of the balancing decision algorithm, and after the balancing decision, we need the task migration module.

C. Task Migration

The task migration module will check if the task value returned during the load decision process is empty at first. If it is empty, the task migration is not needed, and the task is exited directly. Otherwise, it parses the task migration information and obtains the outgoing core ID and the incoming core ID. Finally, the inter-core task migration function is called to complete the final task migration. The task management module implements the inter-core task migration function, so we will not elaborate on it here.

IV. Testing

In this chapter, we design some test cases to test the load balancing method proposed in this paper to verify the effectiveness of it and analyze its performance.

A. Testing Environment

This paper uses the i.MX6Q development board manufactured by Freescale Semiconductor for testing. The i.MX6Q chip integrates four high-performance, low-power ARM Cortex-A9 processor cores [35], which are homogeneous multi-core processor chips. Each core can reach a maximum frequency of 1 GHz and has an independent Neon coprocessor, private timer, watchdog, L1 data cache, and L1 instruction cache of 32 KB in size. Besides, multiple cores share a 1MB-sized L2 cache, SCU, and general-purpose interrupt controller.

The board is equipped with 2GB of DDR3 memory and 8GB of eMMC, and an SD card slot is provided. Since the board is not provided with an IDE, the standard cross-compilation method for embedded development will be used for testing. The kernel code is compiled on the PC with GCC and Make, and then the kernel image is loaded into the board's memory via SD card to run and print debug information using the serial port. In this test, the development board is connected to the PC through a serial adapter cable to obtain debugging information through a terminal emulator like Xshell or Secure CRT.

B. Functional Testing

a) Purpose of the test: The goal of this test is to test whether the load balancing function is working as expected.

b) Test cases: We designed load_balance_test function as a test of the load balancing functionality. The load_balance_test thread is run on CPU$_0$, which creates six threads in turn and assigns them to other cores using the load balancing algorithm according to the load of the other three cores. It is important to note that these six threads have the same priority, and they run without any external output for the convenience of reading debugging information. Their time-slice sizes are 100ms, 50ms, 50ms, 50ms, 50ms, and 50ms. After the other three CPUs receive the threads successfully, they resume their normal operation immediately. In addition, after all six threads are received, CPU$_0$ forces the thread work$_3$ to end and performs dynamic load balancing twice.

c) Test Reuslt: The test records the CPU load changes as shown in TABLE I. When CPU$_0$ creates a new task, it will check the load on the other three cores and select the CPU with the lightest load to migrate the newly created task. At first, as there are no tasks on all three cores, CPU$_0$ assigns work$_1$, work$_2$ and work$_3$ to CPU$_1$, CPU$_2$ and CPU$_3$ respectively. When work$_4$ needs to be assigned, CPU$_0$ assigns work$_4$ to CPU$_2$ because the loads of the three CPUs are 100ms, 50ms, and 50ms, respectively, and loads of CPU$_2$ and CPU$_3$ are the same and less than the load of CPU$_1$. Therefore, CPU$_0$ assigns work$_4$ to CPU$_2$. When work$_5$ needs to be assigned, loads of the three CPUs are 100ms, 100ms, and 50ms, respectively, so work5 will be assigned to CPU$_3$, the core with the lightest load. When work$_6$ needs to be assigned, CPU$_0$ assigns work$_6$ directly to CPU$_1$ because loads of all three cores are the same at this time(100ms). The test result shows that the load balancing function can correctly find the CPU with the lightest load and complete the task assignment.

After all six threads are received, CPU$_0$ first call the function of ending threads in inter-core communication for CPU$_3$ to force the end of work$_3$. At this time, the CPU load is shown in row 4 of TABLE I, and CPU$_0$ will turn on an active load balancing. As is shown in row 5 of the TABLE I, the loads of CPU$_1$, CPU$_2$ and CPU$_3$ are 150ms, 100ms and 50ms respectively. Based on the load decision, CPU$_0$ migrates the work6 with a time slice size of 50ms on CPU$_1$ to CPU$_3$, making the CPU load balanced.

TABLE I
TABLE OF CPU LOAD CHANGES

Step	**CPU LOAD**		
	CPU_1	CPU_2	CPU_3
1	100($work_1$)	50($work_2$)	50($work_3$)
2	100($work_1 1$)	100($work_2$,$work_4$)	50($work_3$)
3	100($work_1$)	100($work_2$,$work_4$)	100($work_3$,$work_5$)
4	150($work_1$,$work_6$)	100($work_2$,$work_4$)	50($work_5$)
5	100($work_1$)	100($work_2$,$work_4$)	100($work_5$,$work_6$)

C. Performance Test

In this section, we perform a series of performance tests on the microkernel with load balancing enabled.

As mentioned before, we tested on the i.MX6Q chip. This chip belongs to the ARMv7-a architecture and provides a Performance Monitoring Unit (PMU). PMU as a hardware performance testing tool dramatically facilitates the testing work and has a higher testing accuracy [36], so we will directly use the PMU module for performance testing. PMU is designed to respond externally to the processor time consumption by counting the processor run cycles. In this paper, the processor's frequency was set to 792 MHz when conducting the test, which means that one cycle takes 1/792 us.

a) Purpose of the test: This test aims to reflect the performance of the load balancing algorithm by testing the execution cycles of the functions get_free_core() and get_transfer_task().

b) Test cases: In the task management module under multi-core, load balancing algorithms must be used to maintain load balancing on multi-core. In this paper, there are three parts of load balancing: load monitoring, balancing decision, and task migration.

The most important of these parts are load monitoring and balancing decisions. From the perspective of load balancing triggers, there are two functions, get_free_core() and get_transfer_task(). The former is called when a new task is created, and the latter is called when load balancing is triggered at timing. Therefore, we reflect the performance of the load balancing algorithm by counting the execution cycles of these two functions.

c) Test Reuslt: The test results are shown in TABLE II and TABLE III. From TABLE II, the average time taken for one get_free_core() function execution is about 4882 cycles, which is about 6.164 µs. From TABLE III, the average time consumed for one get_transfer_task() function execution is about 6596 cycles, which is about 8.328 µs. It can be seen that the timing-triggered load balancing requires more time overhead because it involves more information collection and analysis processing. Nonetheless, their time consumption is in the µs range and has low latency characteristics.

TABLE II
THE TIME CONSUMPTION OF THE GET_FREE_CORE()

The number of tests	Time consumption(Cycles)		
	Total time consumption	*Average Time consumption*	*Maximum time consumption*
10	48857	4885	4950
20	98671	4933	5083
30	147803	4926	5083
40	196760	4919	5083
50	244120	4882	5083

TABLE III
THE TIME CONSUMPTION OF THE GET_TRANSFER_TASK()

The number of tests	Time consumption(Cycles)		
	Total time consumption	*Average Time consumption*	*Maximum time consumption*
10	65828	6582	6625
20	131795	6589	6635
30	197863	6595	6660
40	263770	6594	6660
50	329814	6596	6660

D. System throughput and CPU utilization test

a) Purpose of the test: In multi-core operating systems, tasks can run in parallel on multiple cores, enabling greater throughput. When load balancing is enabled, the overall efficiency can be improved. This test aims to test the improvement of load balancing on system efficiency and CPU utilization.

b) Test cases: In this test case, we perform matrix multiplication to test CPU utilization and system throughput. Firstly, in the multi-core environment without load balancing on four dimensions, 32×32, 64×64, 128×128, and 256×256 are matrix multiplied separately. As shown in Equation 2, the matrix elements are shaped, and each sub-block of the matrix corresponds to a computational task. The matrix is divided into four sub-blocks of equal size, and then sub-block 1 and sub-block 2 are placed on core 1, and each sub-block is repeated 5 times so that there are 5 sub-block tasks 1 and 5 sub-block tasks 2 on core 1. Sub-block 3 and sub-block 4 are placed on core 2 and core 3, respectively, and each sub-block is repeatedly computed 7 times. There are 7 sub-block tasks 3 and 7 sub-block tasks 4 on core 2 and core 3, respectively. Finally, the results are summarized by CPU$_0$.

Secondly, load balancing is turned on, and the initial conditions are the same as before. Core 1 runs 5 task sub-blocks 1 and 5 task sub-blocks 2, core 2 and core 3 run 7 task sub-blocks 3 and 7 task sub-blocks 4 respectively.

$$\begin{aligned} C = AB &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \end{aligned} \quad (2)$$

c) Test Reuslt: After the calculation process is executed 100 times, we record the average cycle time consumption as shown in TABLE IV and Fig. 5. Besides, the CPU utilization in 256 × 256 dimensions is recorded as shown in TABLE V. CPU utilization is the CPU execution time for non-system idle processes / total CPU execution time.

TABLE IV
THE COMPARISON OF AVERAGE CYCLE CONSUMPTION

Matrix Dimension	Average cycle consumption(1000Cycles)	
	load balancing off	*load balancing on*
32×32	78385	78173
64×64	91798	88981
128×128	441455	369128
256×256	32374965	2733432

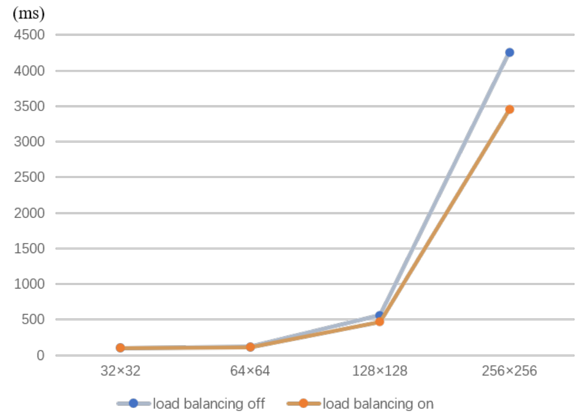

Fig. 5. The comparison of average time consumption

TABLE V
THE CPU UTILIZATION COMPARISON

CPU CORE ID	Load balance off		Load balance on	
	TASK	*CPU utilization*	*TASK*	*CPU utilization*
CPU_0	$IDLE$	0%	Load balance IDLE	0.0082%
CPU_1	5*$TASK_1$ 5*$TASK_2$	100%	4*$TASK_1$ 4*$TASK_2$	100%
CPU_2	7*$TASK_3$	70%	7*$TASK_3$ 1*$TASK_1$	100%
CPU_3	7*$TASK_4$	70%	7*$TASK_4$ 1*$TASK_2$	100%

From the TABLE IV and Fig. 5, we can see that when the matrix dimension is 32×32, the average time consumption does not decrease when load balancing is turned on. This is because we directly assigned tasks to the core in our test, which skips the load balancing triggered at task creation, so only the load balancing triggered at timing exists. In the system, we set up a timed load balancing that is triggered once every 100ms. So, in this case, load balancing will not be triggered because the average time consumption has not reached 100ms. Load balancing will be triggered only if the matrix dimension is further increased and the average time consumption is greater than 100ms.

Before triggering load balancing, the average time consumption is longer because the computing tasks of core 1 are two more than other cores. After load balancing is triggered, the average time consumption is significantly reduced because each CPU core has 8 tasks. And as the matrix dimension increases, the average time consumption grows, and the more pronounced the improvement brought by load balancing. When the matrix dimension is 256×256, the performance improvement is about 20% when load balancing is turned on.

From TABLE V, when load balancing is not enabled, CPU_0 is free and running IDLE tasks, and the utilization rate of CPU_0 is 0%. CPU_1 runs on 10 tasks while CPU_2 and CPU_3 are running on 7 tasks, so CPU_1 is 100% while CPU_2 and CPU_3 are 70%. When load balancing is enabled, CPU_0 is running IDLE tasks and load balance tasks, and the utilization rate of CPU_0 is about 0.0082%, which means the consumption of CPU by load balancing is extremely low and negligible. Meanwhile, CPU_1, CPU_2, and CPU_3 are running 8 tasks on each core after load balancing, and the utilization rate is 100%. Therefore, in this case, the average CPU utilization rate improvement is about 20% after load balancing is enabled.

E. Test Summary

In this chapter, we conducted a series of tests on the load balancing module of the Mginkgo. The test results show that it works well and can perform load balancing when there is a load imbalance in the system so that all processors in the system try to achieve the maximum throughput rate and resource utilization. In addition, the performance overhead that the load balancing module brings to the system was also tested. It shows that the average time spent for load balancing triggered by new task creation is about 4882 cycles(about 6.164 µs). The average time consumption of load balancing triggered by timing is about 6596 cycles(about 8.328 µs). The time consumption of load balancing is in the microsecond range, which means the latency of load balancing is extremely low. Finally, we tested the improvement in system throughput and CPU utilization rate due to load balancing. The results show that load balancing can improve 20% CPU utilization rate under the preset case, and the CPU utilization rate occupied by load balancing is negligibly low at about 0.0082%.

V. CONCLUSION AND FUTURE WORK

To adapt to the trend of multi-core of embedded devices and provide safe, reliable, high performance, and low power consumption embedded systems, we design a centralized global dynamic multi-core load balancing approach. In this paper, the core where the rootserver is located performs the global load balancing regulation. The design of the module is divided into three parts as follows.

- **Load monitoring:** The primary function is to collect the task load on different cores in the system and record it in the structure load_stat_t. Moreover, determine if there is a load imbalance through the information recorded in the structure.
- **Balancing decision:** There are two ways to trigger the balancing decision. One is triggered when a new task is created, and the other is triggered at timings. When a new task is created, based on the results of load monitoring, the core ID of the current system that is relatively idle is obtained, and the task will be migrated there. When triggered at timings, it will analyze the load on each core in the system and get the thread that will be migrated.
- **Task Migration:** Determine whether the output requires task migration of the balancing decision. If task migration is required, parse the task migration information and call the task migration function.

We have implemented the load balancing module on the Mginkgo microkernel and conducted a series of tests on its functionality, performance, and the latency that it brings to the system. The test results show that the time consumption of load balancing is in the microsecond range, which means the latency of load balancing is extremely low. Finally, we set up a load-imbalanced case and tested the system throughout improvement and CPU utilization rate improvement by enabling load balancing. The test results show that load balancing can improve a 20% CPU utilization rate under the preset case.

Although we have implemented the load balancing module on the Mginkgo microkernel, there are still some issues in it, such as whether the threshold parameter T is chosen as the optimal value, whether it would be better to involve the core where the rootserver is located in the scheduling, etc. We hope to research and solve them in our future work.

ACKNOWLEDGMENT

This paper is funded by the project of the state grid corporation of china in 2020 " Integration Technology Research and Prototype Development for High End Controller Chip"(5700-202041264A-0-0-00). The corresponding author is Kun Xiao.

REFERENCES

[1] Moore, G. E. . "Cramming More Components Onto Integrated Circuits." Proceedings of the IEEE 86.1(2002):82-85.
[2] Geer, and D. "Chip makers turn to multicore processors." Computer 38.5(2005):11-13.
[3] Wei, P. , et al. "The New Hardware Development Trend and the Challenges in Data Management and Analysis." Data Science and Engineering 3.12(2018):1-14.
[4] Heiser, G. . "Secure embedded systems need microkernels." (2006).
[5] Katre, K. M. . "Policies for Migration of Real-Time tasks in Embedded Multicore Systems." rtss dec washington d.c.usa (2010).
[6] El-Azab, H. A. , et al. "Design and implementation of multicore support for a highly reliable self-repairing μ-kernel OS." 2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS) IEEE, 2017.
[7] Bo, Z. , et al. "Energy efficient near-threshold chip multi-processing." International Symposium on Low Power Electronics & Design ACM, 2007.
[8] Sawalha, L. H. . Exploiting heterogeneous multicore processors through fine-grained scheduling and low-overhead thread migration.. Diss. The University of Oklahoma. 2012.
[9] ARM. ARM® Cortex ® -A9 MPCore Revision: r4p1 Technical Reference Manual[M]. England: ARM Limited, 2016, 1-13.
[10] Al-Dahoud, et al. "Load Balancing of Distributed Systems Based on Multiple Ant Colonies Optimization." American journal of applied sciences 7.3(2010):428-433.
[11] Koufaty, D. A. , D. Reddy , and S. Hahn . "Bias scheduling in heterogeneous multi-core architectures." European Conference on European Conference on Computer Systems DBLP, 2010.
[12] Craeynest, K Van , et al. "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)." Acm Sigarch Computer Architecture News 40.3(2012):213-224.

[13] Rehman, M. , and M. Asfand-E-Yar . ”Scheduling on Heterogeneous Multi-core Processors Using Stable Matching Algorithm.” International Journal of Advanced Computer Science and Applications 7.6(2016).
[14] Saez, Juan Carlos, et al. ”On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors.” The Computer Journal 61.1 (2018): 74-94.
[15] Tavana, M Khavari , et al. ”ElasticCore: A Dynamic Heterogeneous Platform With Joint Core and Voltage/Frequency Scaling.” IEEE Transactions on Very Large Scale Integration Systems PP.2(2017):1-13.
[16] Saez, J. C. , et al. ”ACFS: a completely fair scheduler for asymmetric single-isa multicore systems.” the 30th Annual ACM Symposium ACM, 2015.
[17] Zheng, S. Q. , and W. Jie . ”Dual of a complete graph as an interconnection network.” Proceedings of SPDP ’96: 8th IEEE Symposium on Parallel and Distributed Processing IEEE, 1996.
[18] Jadon, S. , and R. S. Yadav . ”Load Balancing in Multicore Systems using Heuristics Based Approach.” International Journal of Intelligent Systems and Applications 10.12(2018):56-68.
[19] Baruah, S. . ”Techniques for Multiprocessor Global Schedulability Analysis.” IEEE International Real-time Systems Symposium IEEE, 2007.
[20] J. Luo and N. Jha, “Power-efficient scheduling for heterogeneous distributed real time embedded systems”, IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems, 2007, 1161–1171.
[21] Srinivasan, A. , and J. H. Anderson . ”Optimal rate-based scheduling on multiprocessors.” Journal of Computer & System Sciences 72.6(2002):1094-1117.
[22] J Corbalán, A. Duran , and J. Labarta . ”Dynamic Load Balancing of MPI+OpenMP Applications. ” IEEE Computer Society (2004).
[23] Keren, A. , and A. Barak . ”Adaptive placement of parallel Java agents in a scalable computing cluster.” Concurrency & Computation Practice & Experience 10.11-13(2010):971-976.
[24] R. Mohan and N. P. Gopalan, ”Dynamic Load Balancing using Graphics Processors”, International Journal of Intelligent Systems and Applications (IJISA), Vol.6, No.5, pp.70-75, 2014. DOI: 10.5815/ijisa.2014.05.07.
[25] Verma, M. , N. Bhardwaj , and A. K. Yadav . ”Real Time Efficient Scheduling Algorithm for Load Balancing in Fog Computing Environment.” International Journal of Information Technology and Computer Science 8.4(2016):1-10.
[26] P. S. Kshirsagar and A. M. Pujar, ”Resource Allocation Strategy with Lease Policy and Dynamic Load Balancing”, International Journal of Modern Education and Computer Science (IJMECS), Vol.9, No.2, pp.27-33, 2017. DOI: 10.5815/ijmecs.2017.02.03.
[27] Yang, H. , X Zhang, and J. Zhao . ”A Dynamic Feedback-based Load Balancing Methodology.” Journal of Changchun University of Science and Technology(Natural Science Edition) (2017).
[28] D Shelepov, et al. ”HASS: a scheduler for heterogeneous multicore systems.” ACM SIGOPS Operating Systems Review 43.2(2009):66-75.
[29] Kang, J. , and D. G. Waddington . ”Load Balancing Aware Real-Time Task Partitioning in Multicore Systems.” IEEE IEEE, 2012:404-407.
[30] Wang, T. , et al. ”Fair scheduling on dynamic heterogeneous chip multiprocessor.” (2014).
[31] Liu, G. , J. Park , and D. Marculescu . ”Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems.” 2013 IEEE 31st International Conference on Computer Design (ICCD) IEEE, 2013.
[32] Becchi M, Crowley P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proc. of the 3rd Conf. on Computing Frotiers. 2006. 29
[33] Li T, Baumberger D, Koufaty DA, et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In: Proc. of the 2007 ACM/IEEE Conf. on Supercomputing. 2007. 53.
[34] Jadon, S. , and R. S. Yadav . ”Deadline-Constrained Tasks’ Scheduling in Multi-core Systems Using Harmonic-Aware Load Balancing.” Arabian Journal for Science and Engineering (2020).
[35] NXP. i.MX 6 Series Application Processors Fact Sheet[M]. Texas: NXP, 2015, 1-2.
[36] Eranian, S. . ”The perfmon2 interface specification.” (2009).