Inter-Core Communication Mechanisms for Microkernel Operating System based on Signal Transmission and Shared Memory

Cheng Liu
School of Computer Science and Engineering, University of Electronic Science and Technology of China
Chengdu, China
chengliu_uestc@163.com

Lei Luo
School of Computer Science and Engineering, University of Electronic Science and Technology of China
Chengdu, China
lluo@uestc.edu.cn

Meng Li
School of Computer Science and Engineering, University of Electronic Science and Technology of China
Chengdu, China
horimon@163.com

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

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

2021 7th International Symposium on System and Software Reliability (ISSSR) | 978-1-6654-3431-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSSR53171.2021.00031

Abstract—With the coming of the Internet of things(IoT) era and the development of semiconductor equipment, multi-core processors have begun to be widely used in IoT devices to meet their requirements for powerful processing capabilities. Unlike desktop or server operating systems such as Linux, current embedded operating systems often do not support multi-core processors well. Tasks on different cores often require infor-mation exchange, known as inter-core communication, which significantly impacts the processing performance of multi-core operation systems. In this paper, we proposed an inter-core communication method based on signal transmission and shared memory, which is flexible and various types of data can be trans-ferred efficiently. We have implemented and experimented with it on our own microkernel operating system named Mginkgo. The experimental results show that the average time to trigger an inter-core interrupt is about 0.093 microseconds. The average inter-core interrupt processing time is about 3.986 microseconds. And the communication time of the system for multi-core Inter-Process Communication(IPC) is about 18us, which is the same as that of single-core IPC. The inter-core communication method proposed in this paper achieves very low latency with almost no performance consumption and maintains the high performance of the whole system.

Keywords: inter-core communication, multicore, microker-nel

I. Introduction

With the continuous development of the chip manufacturing industry, single-core processors cannot meet the needs of the growing market in terms of performance, power consumption, and price/performance ratio. The performance of the processor grows with the increase in the number of transistors per unit area on the chip [1], which is the law pointed out by Moore's law, but the processor will also be limited by heat, clock frequency, etc. Therefore, multi-core processors came into view and started to flourish. With the development of multi-core technology and the advent of the IoT era, more and more embedded devices are adopting multi-core processors.

On a multiprocessor chip, each CPU shares not only devices such as memory and I/O, but also L2 cache, front-end bus, and memory, which are connected via a high-speed bus. In this way, CPU cores can communicate with each other using the high-speed bus, which significantly increases the speed of communication between tasks running in parallel. In addition, with the increasing number of cores, the chip area is smaller in a multi-core processor compared to a multiprocessor chip because there are no excessive buses and L2 cache. There are also some problems with multi-core processors. For example, when L2 Cache and the front-end bus become shared resources, there will be more resource contention problems. Although multi-core processors need to solve more shared resource contention problems, the advantages of its simple architecture, small chip area, and fast inter-core communication are more important. So it has become a more mainstream processor development direction [2]. Therefore, the multi-core system will be more and more applied in the embedded field [3]. The research of operating systems for multiprocessor environments has become particularly critical and is rapidly becoming a research hotspot in the industry [4].

This work is supported by the Introducing Program of Dongguan for Leading Talents in Innovation and Entrepreneur (Dongren Han [2018], No. 738).

978-1-6654-3431-7/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSSR53171.2021.00031

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

Currently, there are three main common kernel architecture implementations: monolithic kernel, microkernel, and a combined form of both also known as hybrid kernel [5]. The monolithic kernel was the first kernel to emerge with a hierarchical structure that runs all system base services such as thread management, memory management, interrupt handling, I/O communication, and the file system in the kernel. Because the monolithic kernel runs a large number of non-essential service processes in kernel space, it exposed many avenues of attack and had poor security. At the same time, the monolithic kernel has too much code that cannot be pruned, making it difficult to meet the requirements of using relatively few resources on embedded devices. To overcome the shortcomings of the monolithic kernel, the microkernel only provides the most basic kernel services such as task management, inter-task communication, and interrupt management. The microkernel significantly reduces the amount of kernel code and security risks. The current multi-core operating systems widely used on embedded platforms in the market mainly include Android, IOS, etc. Among them, Android OS, which originated from Linux, is an open-source project launched by Google and is widely used on consumer electronic devices such as smartphones, tablets, and TVs after customization by domestic and foreign smart device manufacturers.

As we can see, these operating systems are designed as desktop operating systems, and most of the kernels are monolithic kernel designs. As the functionality of these kernels continues to increase, the amount of kernel modules and code inevitably increases. There are many unpredictable vulnerabilities, and the kernel is subject to significant security risks. The microkernel is very suitable for embedded platforms. The microkernel is considered to be the most mainstream operating system for embedded platforms in the future [6] for its lightweight and security. However, most of the microkernel operating systems today do not provide good support for multi-core processor environments, which makes it difficult to bring out the best performance of the hardware of embedded platforms with multicore. Therefore, research on multi-core support for microkernel operating systems is crucial for the future development of the embedded field.

The performance of a multi-core operating system does not simply increase with the number of cores [7], this is because the performance of a multi-core operating system can be affected by many other factors. For example, the exchange of information about the tasks running on different cores, how the operating system running on the multi-core processor manages the cores, how the software running on the operating system is designed to be used, and so on [8]. The design of how information is passed between different cores, which is the inter-core communication, is a critical factor that affects the performance of a multi-core operating system. It affects not only the interaction between the tasks running on different cores, but also the scheduling of the operating systems, the overall performance of the system, and the throughput, etc.

However, in the current market, different multi-core processor platforms usually provide different inter-core communication mechanisms. The complexity and diversity of inter-core interrupt mechanisms and the lack of a unified standard bring great trouble to software developers. A unified, portable, and efficient inter-core communication mechanism would significantly reduce these difficulties, but such a solution does not yet exist [9]. A fast inter-core communication mechanism provides high-performance parallel programs with the transfer of shared data between tasks of different cores [10]. Therefore research on inter-core interrupt mechanisms on multi-core operating systems is a further need for current and future market research. In a multi-core operating system, how to solve the problem of mutual exclusion of data transmission and task synchronization on different cores is the key to realize efficient communication between cores.

In this work, we propose an inter-core communication mechanism based on signal transmission and shared memory. We implement this mechanism on the i.MX6Q chip platform on our own microkernel operatingsystem named Mginkgo. We also design some test cases to test this mechanism. The test results show that the average time to trigger an inter-core interrupt is about 0.093 microseconds. The average time to process an inter-core interrupt is about 3.986 microseconds. And the communication time of multi-core IPC is about 18us, which is the same as that of single-core IPC. The inter-core interrupt method proposed in this paper achieves very low latency with almost no performance consumption and maintains the high performance of the whole system.

The structure of this paper is organized as described below. The second section describes the related research work of other researchers on multi-core and inter-core communication techniques. The third section describes in detail the design and implementation of our inter-core communication mechanism. The fourth section introduces the design of test case and test results on the performance of our designed inter-core communication mechanism and microkernel. The fifth section summarizes the paper and concludes.

II. Related Work

This section is divided into three subsections. Section A describes two different operating modes of a multi-core operating system. Section B introduces the design and features of the microkernel named Mginkgo developed by our team. Section C describes the design of inter-core communication mechanisms proposed by other researchers under different operating modes and analyzes their proposed inter-core communication mechanisms. Then a preliminary description of the inter-core communication mechanism designed and implemented in this paper is presented.

A. Multi-Core Operating System

The operating system is the first layer of software running on top of the hardware, which is responsible for managing the hardware and software resources of the computer. The current multi-core operating systems are divided into two main modes of operation, symmetric multi-processing(SMP) mode and asymmetric multi-processing(AMP) mode [11].

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

In the AMP model, there are multiple OS instances running on one or more processor cores in the whole system. Each of these running OS instances has its own independent resources, such as memory, peripherals, etc. These resources are generally configured by the user during the boot phase and are not changed after booting. Since each of these OS instances has its own independent resources, they communicate with each other by dividing a block of memory from the shared memory. Operating system instances running on different cores are generally not equal. For example, in Rami Matarneh's operating system designed for multi-core architectures, OS instances running on slave cores must be authorized by the master core to run system services [12]. Operating system schedulers usually assume identical hardware. Thus, asymmetric multi-core processors pose a unique challenge to the kernel. While scheduling threads in a "good" asymmetric-aware way can bring the benefits of heterogeneity, a "bad" way can lose all the benefits of heterogeneity, as well as the benefits of using multiple cores [13]. Balakrishnan et al. [14] showed that an asymmetric imperceptive scheduler leads not only to poor application performance but also to application instability. Wei Zhang [15] proposed two possible models for extending the embedded real-time operating system RODOS to a multi-core version: the SMP model and the AMP model. He implemented a working version of the AMP model. It is able to run tasks on a multi-core platform, provides communication and synchronization API to avoid deadlocks, and allows parallel applications to share data and messages in a secure way. Instead of using a shared pointer to hold data in both directions, its inter-core communication is done in such a way that each of the two pointers acts as a one-way channel dedicated to one core to avoid contention when two cores try to send data to opposite cores.

In the SMP mode, there is only one OS instance in the whole system, which controls all the resources of the multi-core processor, such as memory and peripherals, and they are shared as shared resources among all cores. In this mode, a design idea similar to that of homogeneous multi-core processors is used, i.e., the processor cores are simply used as a computational resource and only need to perform the tasks assigned to them. The cores have equal status and share resources with each other. Compared with AMP mode, SMP mode operating system has the advantages of simple structure, high communication efficiency, and convenient load balancing, which can better exploit the performance and power advantages of the multi-core processor platform. There are many operating systems on the market that can support SMP mode, such as Linux, Windows, and VxWorks, etc. Hongxing Wei et al. [16] implemented RT-ROS, which provides an integrated real-time/non-real-time task execution environment, so real-time and non-real-time ROS nodes can run on different processor cores respectively real-time operating systems or on Linux operating systems. This demonstrates that multi-core operating systems can effectively provide high-performance real-time support for multi-core processor platforms by exploring multi-core architectures. Daniel Cederman [17] improved and extended SMP support for RTEMS real-time operating systems. He developed and implemented a multi-core task management API, and demonstrated the feasibility and usefulness of multi-core processors for spatial payload software. Adhiraj Joshi et al. [18] proposed Twin-Linux, a technical solution that opens the door of hybrid computing for x86 and open source communities. When a hardware box needs to cater to multiple compute classes, the scheme runs different copies of the kernel simultaneously on different cores of a multi-core system and uses IPIs (Inter Processor Interrupts) and shared memory to provide synchronization between the cores.

The SMP mode has a simple structure, so it is more suitable for embedded systems. It can take advantage of the performance and power consumption of current multi-core processors. In this paper, we design an inter-core communication mechanism suitable for SMP mode. The design of this mechanism makes full use of the features of resource sharing and equal status of cores in SMP mode.

B. Mginkgo Microkernel

The Mginkgo microkernel is designed and implemented by our team based on the third-generation microkernel seL4, which is designed to meet the increasing requirements of the industry in terms of security, real-time, and reliability of operating systems for industrial control.

The core functional modules of the microkernel consist of five major functional modules: the capability subsystem, task management, memory management, IPC module and interrupt management. The capability subsystem is used to record the access permissions of tasks to kernel resources and is responsible for checking the permissions of tasks when they access resources. The task management module is responsible for the organization and scheduling of all threads in the system, and the IPC module implements the communication mechanism between tasks. The inter-core communication mechanism mainly requires memory management and interrupt management, and the multi-core IPC is also affected by inter-core communication. The following is an expansion of the memory management, interrupt management, and IPC modules.

Memory management is responsible for the management of memory resources and the conversion between physical and virtual addresses. For memory resource management, the kernel divides the memory resources in the platform into general memory and kernel-specific memory and uses red-black trees to organize free memory blocks. For general memory, the kernel gives the control authority to the root service, so the allocation and release of memory used by applications is done by the root service running in user space, and the kernel only checks the general memory for out-of-bounds or permission errors. At the same time, the kernel's data and some key data structures such as task lists, page tables, and IPC Buffer are stored in kernel-specific memory, which is only available to the kernel. Finally, for the address translation, the kernel uses a two-level page and the Memory Management Unit (MMU) provided by the hardware platform

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

to implement the virtual to physical address translation and permission checking.

The interrupt management module is responsible for managing interrupt resources. It also provides interrupt registration, notification, response, enable and disable functions. In the kernel, interrupts are divided into exception handling, hardware interrupts, and soft interrupts. At the same time, the handling of interrupts is differentiated according to their usage. If the interrupt is used by the kernel, only the relevant handler function needs to be called in the kernel state. If the interrupt is used by the application, then it runs in the user state and the kernel cannot execute the code directly. Therefore, in this case, the kernel treats the interrupt triggered by the application as an event and reuses the Notification structure in the IPC module to preallocate a Notification object for each interrupt, storing the interrupt handler and indicating whether the interrupt is currently registered, belongs to the kernel or to the application, etc.

The IPC module, also known as the inter-task communication module, implements two basic types of communication between tasks. One is the event notification mechanism, which implements blocking and non-blocking wait and notification, respectively; the other is message passing, which also implements blocking and non-blocking send and receives, respectively. They rely on two kernel objects, Notification and Endpoint, to implement them respectively. For event notification, the wait and occurrence of an event are reflected in the change of the Notification's state, thus completing the change of the task's execution state, without the need for data copying. For messaging, communication is only possible when two tasks get the same Endpoint object. Meanwhile, each task has an IPC Buffer buffer, and the message passing mechanism copies the contents of the IPC Buffer of the "sending task" to the IPC Buffer buffer of the "receiving task" and activates the latter to complete the message passing. The message delivery mechanism completes the message delivery by copying the contents of the IPC Buffer of the sending task to the IPC Buffer buffer of the receiving task and activating the latter.

The microkernel runs stably on single-core processors, but there is no support for multi-core platforms yet. In this paper, we investigate the most important inter-core communication mechanism in the multicore extension of this microkernel.

C. Inter-Core Communications

The inter-core communication approaches nowadays applied to multi-core operating systems can be divided into two categories [19]. One type of approach is the virtualization approach, which adds an intermediate layer between the hardware and the operating system to shield the operating system from the hardware, and uses this intermediate layer to manage various resources and provide a standard interface for the operating system to use these resources. Operating systems on different kernels running on this middle layer will consider they are communicating with other machines when they communicate with each other. Another way to communicate is through shared memory and the IPC message queue. The OS instance that sends the message first writes the contents of the communication message to the message queue in shared memory and then triggers an inter-core interrupt. The OS running on another core waiting for the message starts the receiving program by receiving the inter-core interrupt, and the receiving program retrieves the communication contents from the message queue in shared memory.

The virtualization method is difficult to be applied in practice because of its disadvantages such as poor real-time performance, the heavy burden of system resources, and difficult implementation [20]. Therefore, the industry usually adopts the second approach to implement the inter-core communication mechanism.

There are many ways to implement inter-core communication. How to choose the implementation depends on the size of the message to be communicated between the cores. If the message is small, the last-level cache can be used to deliver the communication message. Micaiah Chisholm et al. [21] proposed a mechanism for inter-core communication on multiprocessor platforms through cache isolation and DRAM banks, but this mechanism requires multiple cores to communicate on the same DRAM banks, which limits the size of the communication content and its communication speed is slow. This mechanism cannot be used when delivering large inter-core communication messages. Rohan Tabish et al. [22] proposed a communication core model (CCM) which enables inter-core communication by limiting the amount of inter-core interference in partitioned multi-core systems. This model avoids the problem of slow communication by minimizing the number of cores accessing DRAM banks at any point in time. Konstantina Mitropoulou et al. [10] proposed a new Single-producer / single-consumer queue design that provides fast inter-core communication even at the fine granularity and develops fast inter-kernel communication and developed Lynx, a new architecture that exploits existing processor hardware and OS support for exception handling to minimize the overhead of in- and out-queue operations. This design optimizes the message passing process for message queues in shared memory. Yang Nie et al. [7] implemented a shared memory-based inter-core communication mechanism in Zynq-7000. The shared memory of this mechanism is divided into OCM (On-chip memory) and DDR memory. It has been experimentally demonstrated that OCM provides very high performance and low latency access from both processors compared to DDR memory. Zhiyi Yu et al. [23] demonstrated that a hybrid communication mechanism supporting both message passing and shared memory can provide higher performance and energy efficiency. It has been experimentally verified that this mechanism improves the throughput rate by nearly 2.5× compared to the traditional message passing architecture and the normalized decoding efficiency by 3.7×. Zhiyi Yu et al. [24] proposed a 16-core processor with a hybrid inter-core communication scheme using shared memory and message passing. A two-dimensional on-chip mesh network (NoC) is used to support message-passing communication. Also, a cluster-based memory hierarchy containing shared memory

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

supports shared memory communication, a hardware-assisted mailbox inter-core synchronization method to support inter-core communication, and a new memory hierarchy to achieve higher energy efficiency. Xiaojie Xu et al. [25] proposed a shared-memory inter-core communication model (TASMCM) based task allocation to reduce mode-to-mode communication latency, ensures consistent data synchronization, and improves the capability of multi-core processor parallel processing. The design specifically targets the communication mechanism for high parallelism application performance, but if the application has low parallelism, then TASMCM will increase the time overhead.

The schemes proposed by the above researchers implement inter-core communication based on dividing shared memory. The design of the message transmission mechanism is less flexible and the size of the shared memory is deterministic. Secondly, different messages need to divide different data types to deliver the messages. Finally, some of the design ideas are so complex that it is difficult to implement the inter-core communication framework defined by the idea in practical development, which lacks practical feasibility. In this paper, we propose an inter-core communication mechanism based on signal transmission and shared memory to overcome the shortcomings of the above schemes. Our proposed inter-core communication mechanism divides the inter-core communication into two parts: the first part is the passing of control signals, called signal transmission, and the second part is the transferring of specific data, called data communication. Signal transmission refers to the passing of some control signals such as core sleep, core wake-up, rescheduling, etc. between cores to serve as a notification and does not involve any data flow between cores. On the other hand, data communication involves the exchange of data on different cores. The amount of data involved is huge. The size of shared memory in the design of the inter-core communication mechanism proposed above is determined, so it is very inflexible. By leaving the actual processing part to the interrupt processing function, we use signaling to overcome the inflexibility of the message transmission method and eliminate the need to divide different data types to deliver different messages. This also solves the disadvantage of requiring a fixed size of shared memory. Secondly, the method is divided into two well-defined parts, which reduces the difficulty of development and better practicality exists.

III. Our Work

A. The Design of Inter-Core Communication

In the existing IPC module of mginkgo microkernel, the message passing and event notification mechanisms are implemented through two kernel objects, Notification and Endpoint, which provide a safe and reliable mechanism for communication between tasks in the microkernel. In a multi-core environment, different cores often need to collaborate and communicate with each other. The existing IPC module is not enough to complete the message communication between cores, and an inter-core communication module needs to be designed and implemented to complete the above functions.

The inter-core communication design in this work is divided into two parts, signal transmission and data communication, which work together to complete the inter-core communication. The design ideas for each of the two parts are described next.

First is the design idea of signal transmission. As mentioned above, the current signals for signaling are mainly three kinds of control signals: core sleep, core wake-up, and rescheduling. For signal transmission, strong real-time is often required. After the signal is sent, the receiver needs to respond immediately and complete the switching of its operation state according to the signal type, which is done by inter-core interrupt in this paper. The sender of the signal passes the signal to the receiver by triggering a Software Generated Interrupt(SGI), and the current running state of the receiver is immediately interrupted by the signal, and then the corresponding inter-core interrupt processing function is running. Compared to the eight SGI interrupt numbers used in the Linux kernel for inter-core signalings, such as core sleep, core wakeup, time interrupt, and rescheduling, only three interrupt numbers are currently used in this microkernel as needed, for core sleep, wakeup (rescheduling), and one inter-core interrupt dedicated to data communication. Other interrupt numbers are reserved for the microkernel to apply the TrustZone technology. While signaling transmission can change the operational state of other cores, it cannot satisfy all inter-core communication scenarios. For example, when tasks are running together on different cores, one of the tasks needs to send its results to the task on the other core after it has finished running. In this case, not only inter-core interrupts need to be used to complete the notification, but also to complete the specific data transfer work.

On a multi-core processor platform, shared memory refers to an area of memory that is accessible to different CPU cores. When a task on one core modifies this memory, tasks on other cores have access to its changes. Also, since both communicating parties are directly accessing data to the same memory area, the process can be completed quickly even if the amount of communicated data is large. It is thus clear that inter-core interrupts are the basis for inter-core communication, while the shared memory mechanism is the key.

As can be seen in Fig. 1, when thread 0 wants to send data to thread 1 on another core, it needs to store the data in the shared memory first and then notify thread 1 by means of an inter-core interrupt. Thread 1 will read the data from the shared memory and empty its contents after receiving the notification. Of course, since this memory is shared by different cores, the division of this memory will be done during the system startup phase and a special data structure will be set up to facilitate access to the communication data. Finally, synchronous access to this shared resource by different threads is also ensured with the help of inter-core synchronization and mutual exclusion mechanisms.

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

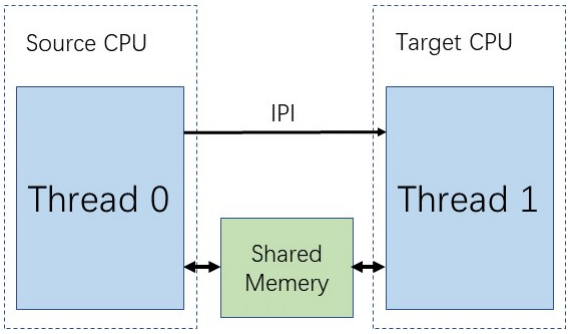

Fig. 1. inter-core data communication

B. The Implement of Inter-Core Communication

In ARM's multi-core architecture, the Generic Interrupt Controller (GIC) is introduced to handle interrupts, which supports interrupt priority setting, interrupt preemption, and inter-core routing of interrupts. the GIC is divided into two parts: the Interrupt Distributor and the CPU Interface [26]. The former is used to trigger and distribute interrupts, while the latter converts the interrupt signals distributed by the former into physical interrupt signals to the CPU to which it is connected.

As mentioned earlier, according to the actual needs of the current microkernel, three inter-core interrupt variables are defined, IPI_RUN_IDLE and IPI_RE_SCHEDULE for signal communication, and IPI_ICC for data communication, which occupies SGI interrupt numbers 8∼10.IPI_RUN_IDLE is used to notify the target CPU to enter the sleep state, that is, to execute the Idle task all the time, while IPI_RE_SCHEDULE is used to notify the target CPU to perform a rescheduling, or to wake up the CPU in the sleep state. Upon receiving an inter-core interrupt, the receiver looks for the information passed to it and then completes the appropriate processing based on the specific data.

The specific process of registration, triggering, and processing of inter-core interrupts can be divided into seven steps as shown in Fig. 2.

1) Inter-core interrupt registration. The setup of the interrupt vector table is completed during the system initialization phase. This process associates each defined inter-core interrupt with the corresponding interrupt handler function to facilitate the call of the corresponding function when an inter-core interrupt is received.
2) Inter-core interrupts triggering. This step uses software triggering to set the SGIR register according to the target CPU serial number, interrupt number, and other information to trigger an inter-core interrupt.
3) Inter-core interrupts convey. This phase is mainly done through hardware, where the Distributor in the GIC distributes the inter-core interrupts to the corresponding cores. The CPU Interface on the target core will check whether it needs to intercept the interrupt based on its information. If an interception is required, this interrupt signal is blocked; if it is allowed to pass, a corresponding physical signal is sent to the CPU's IRQ signal line. After the CPU senses, this interrupt signal, the process of interrupt to convey is complete.
4) Interrupt processing preparation phase. This phase is similar to the preparation work before the general soft interrupt processing, including saving the CPU context information, finding the interrupt processing function, and jumping to the corresponding address to start the execution.
5) Inter-core interrupt handling. Run the interrupt handling function queried in phase 4. This process temporarily blocks all interrupt requests so that the execution of the interrupt handling function is not interrupted.
6) Inter-core interrupts confirm. After the interrupt is executed, the target CPU informs the CPU Interface that it has finished processing the inter-core interrupt by setting the ICCEIR register.
7) Restore Context. Restores the context information saved in stage e above, thus restoring the CPU state before the interrupt occurred.

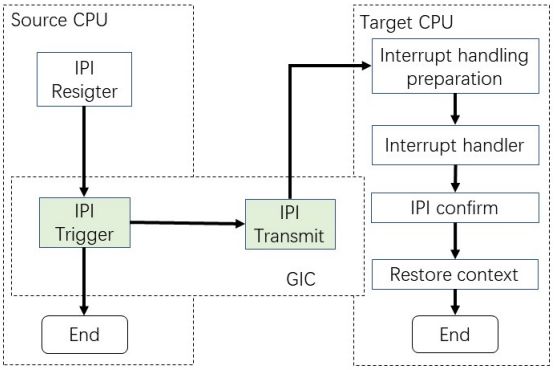

Fig. 2. Inter-core interruption flow chart

Since inter-core interrupts can only provide signal notification, they cannot be used for situations involving data communication such as task migration and function calls. Therefore, for data communication, the microkernel reserves a memory area for each core and provides the ipi_data_t data structure for easy access to communication data, as shown in Fig. 3.

```
typedef struct __ipi_data_t {
    spinlock_t lock;
    unsigned short state;
    unsigned int target_cpu;
    tcb_t* task;
    handler_t handler;
} ipi_data_t;
```

Fig. 3. Ipi_data_t data structure

To improve communication efficiency, each CPU has a

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

copy of its ipi_data_t data structure in which it stores the information attached to the data communication sent to it. One of the variables, lock, is a spinlock to guarantee the mutually exclusive operation of this data structure between cores. The state is used to specify its state, and its optional values are shown in TABLE I below. Also, only when the state is IPIDATA_NOT_SET state, which means the current ipi_data_t is not set, the thread can go to seize the ipi_data_t mutex lock; otherwise, the thread can only spin and wait until the value of state returns to IPIDATA_NOT_SET before it can start seizing.

TABLE I
IPI_DATA_T DATA STRUCTURE STATUS TABLE

State	Data communication function
IPIDATA_NOT_SET	The ipi_data_t variable does not currently hold specific data and is allowed to be set.
IPIDATA_TASK_IN	Notify the target CPU to accept the migrated task.
IPIDATA_TASK_OUT	Notifies the target CPU to migrate the specified task to another CPU.
IPIDATA_TASK_KILL	Notifies the target CPU to terminate the specified thread.
IPIDATA_CALL_FUNC	Notifies the target CPU to execute a function immediately.
IPIDATA_TASK_IPC	Notify the target CPU to make an IPC communication.

The variables in the ipi_data_t data structure, except for the lock and state variables, are set for specific scenarios and uses. For example, target_cpu is used to specify the target CPU. The task is used to specify the target task to be migrated during task migration and to specify the task to be forced to end during task termination. The last handler member is specifically used to specify the code of that function to be called when the state is IPIDATA_CALL_FUNC.

Therefore, compared to the workflow of inter-core interrupts, when there is a need for data exchange between threads on different cores, the initiator of inter-core communication needs to complete the setup of the ipi_data_t structure before triggering the inter-core interrupt. The receiver of inter-core communication needs to read the ipi_data_t structure before jumping to the specific inter-core interrupt processing function and executing it. The process is roughly shown in Fig. 4.

The design and implementation of inter-core communication is then complete. Next are the test case design and test results on the performance of inter-core communication and the performance of the microkernel with the addition of this mechanism.

IV. EXPERIMENTAL TESTING AND ANALYSIS

In this chapter, several test cases are designed to test the implemented inter-core communication mechanism and the microkernel multi-core IPC to verify its correct functionality and to analyze its performance.

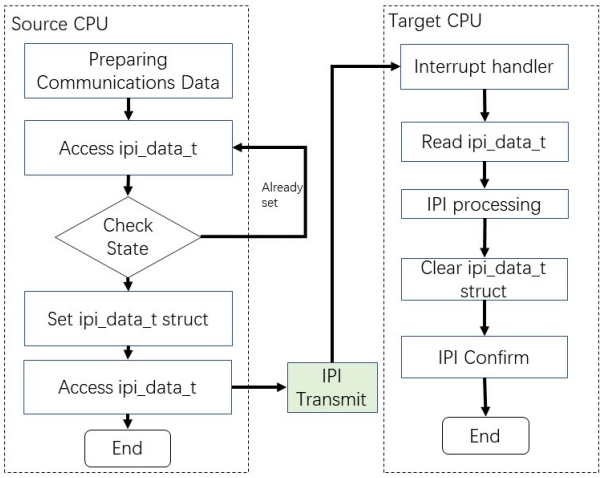

Fig. 4. Ipi_data_t data structure

A. Experimental Test Environment

This paper uses the i.MX6Q development board manufactured by Freescale Semiconductor as a test environment. The i.MX6Q chip used in this development board integrates four high-performance, low-power ARM Cortex-A9 processor cores, which are homogeneous multi-core processor chips. Each core can reach a maximum frequency of 1GHz and has an independent Neon coprocessor, private timer, and watchdog, as well as L1 data cache and L1 instruction cache of 32KB in size. At the same time, multiple cores share a 1MB L2 cache, Snoopy Control Unit (SCU), and a general-purpose interrupt controller.

In addition, the board is equipped with 2GB DDR3 memory and 8GB eMMC, and an SD card slot is provided. Since the board does not provide an IDE, the test will use the common cross-compilation method for embedded development, compiling the kernel code on the PC with GCC and Make, loading the kernel image into the memory of the board via SD card, and printing debugging information using the serial port. In this test, the development board is connected to the PC through a serial adapter cable, so that debugging information can be obtained through a terminal emulator like Xshell or Secure CRT.

B. Functional Test

In the inter-core communication module, signal transmission and data communication are implemented by using inter-core interrupts and shared memory. Among them, the signal transmission provides the functions of CPU sleep and CPU wakeup (rescheduling); data communication provides the five major functions of task move-in, task move-out, task end, function call, and IPC communication. Therefore, this subsection tests them using test cases 4-1, 4-2, 4-3, 4-4, 4-5, and 4-6. They are shown below.

Test Case 4-1: Signal communication of CPU sleep and wake-up test. Run the icc_signal_test thread on CPU0, which will interrupt and resume the normal operation of CPU1

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

successively during the running process. And CPU1 has only two threads, the work1 thread, and the Idle thread. The work1 thread outputs a string every time it is executed, while the Idle thread does not have any output. In addition, the threads running on other CPUs do not affect the normal operation of the two CPUs and their threads.

Test purpose: To test the correctness of signal communication in the inter-core communication module and to observe whether the CPU sleep and wake-up functions provided by it are normal.

Test results: The core sleep and wake-up functions provided by the signal transmission are functioning properly.

Test cases 4-2, 4-3, 4-4, 4-5, and 4-6 are similar to the test procedure and test results of test cases 4-1. The signal transmission and data communication provided in the inter-core communication module is implemented correctly, and the functions provided in the inter-core communication module such as task move-in, task move-out, task end, function call, and IPC communication can work properly.

C. Performance Test

As mentioned earlier, this paper is conducted on the i.MX6Q chip, which is part of the ARMv7-a architecture and provides a Performance Monitoring Unit (PMU), a hardware-level performance test tool that not only facilitates the testing process but also provides higher testing accuracy. Therefore, this paper will directly use the PMU module to perform performance tests. PMU is used to externally reflect the processor time consumption by counting the processor run cycles, and in this paper, the processor frequency is set to 792 MHz for the test, which means that one cycle takes 1/792 microseconds.

First of all, the inter-core interrupt is an essential part of the inter-core communication function module, and its performance will have a great impact on the system, so it is necessary to test the triggering and processing speed of the inter-core interrupt. In this paper, the gic_sgi_trigger() function is used to complete the setting of SGIR registers to trigger inter-core interrupts by means of software. Therefore, this test reflects the performance of inter-core interrupt triggering by detecting the number of cycles consumed during the execution of this function. The test results are shown in TABLE II. The average consumption of triggering an inter-core interrupt is about 74 cycles, which is about 0.093 microseconds. This shows that inter-core interrupts are triggered very quickly and can provide a fast notification.

TABLE II
CYCLE CONSUMPTION TRIGGERED BY INTER-CORE INTERRUPT

Number of test	The total time consuming (Cycles)	The average time consuming (Cycles)	Maximum time consuming (Cycles)
10	547	54	62
20	1144	57	74
30	1742	58	74
40	2334	58	74
50	2911	58	74

Of course, in addition to triggering inter-core interrupts, their handling is also very critical. In this paper, the interrupt handling of the kernel consists of finding the interrupt handler function based on the interrupt vector table, running the interrupt handler function, and the inter-core interrupt confirm, which is the irq_traps_entry() function. Therefore, this test reflects the performance of the inter-core interrupt handling by examining the cycle consumption of this function when it is executed. The test results are shown in TABLE III, from which it can be found that the average time consumed for one inter-core interrupt processing is about 3157 cycles, which is about 3.986 microseconds.

TABLE III
CYCLE CONSUMPTION OF INTER-CORE INTERRUPT PROCESSING

Number of test	The total time consuming (Cycles)	The average time consuming (Cycles)	Maximum time consuming (Cycles)
10	31708	3170	3247
20	63324	3166	3480
30	94906	3163	3480
40	126254	3156	3480
50	157878	3157	3480

Finally, due to the addition of the inter-core interrupt module, the multi-core IPC also needs to communicate between processes of different cores through inter-core interrupts, so the multi-core IPC will inevitably be affected by inter-core interrupts. To test the impact of inter-core communication on multicore IPC, this paper designs test cases to test both single-core IPC and multi-core IPC. The test case consists of creating two threads, one as a sending thread and the other as a receiving thread. The receive thread, after setting the associated receive buffer and timer, will suspend itself and the state will switch to IPC receive. The transmitting thread, will prepare 256 bytes of data, set the timer, and then call the IPC transmission interface to send the 256 bytes of data to the receiving thread. After the data is sent, the receiving thread is woken up, and the receiving thread receives the 256 bytes of data, then stops the timer and calculates the time consumed for this IPC.

This test reflects the impact of inter-core interruptions on multi-core IPC by calculating the execution time of IPC when different threads are in the same core or different cores, and thus judges the performance of multi-core IPC. The results are shown in TABLE IV and TABLE V.

TABLE IV shows the performance test of single-core IPC.TABLE V shows the performance test of multi-core IPC, the average time consumed by a single-core IPC is fast at around 18us, while the average time consumed by multi-core IPC is also around 18us. It can be seen that the average time consumed by the multi-core IPC is relatively consistent with that of the multi-core IPC and does not have a very significant impact on the performance. Therefore, it can be concluded that our design of inter-core communication does not degrade the performance of the system with the necessary functions completed, and its design is relatively reasonable.

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
IPC PERFORMANCE TEST OF SINGLE CORE

Number of test	The total time consuming (us)	The average time consuming (us)	Maximum time consuming (us)
10	184	18	21
20	366	18	21
30	548	18	21
40	731	18	21
50	911	18	21

TABLE V
IPC PERFORMANCE TEST OF MULTI CORE

Number of test	The total time consuming (us)	The average time consuming (us)	Maximum time consuming (us)
10	190	19	26
20	371	18	26
30	559	18	26
40	746	18	26
50	930	18	26

D. Test Summary

In this section, we describe in detail the test design and testing process for the inter-core interrupt mechanism designed and implemented in this paper. The inter-core interrupt mechanism is evaluated from three aspects: test environment, functional test, and performance test, respectively. Through the comparison and analysis of the tests, the inter-core interrupt mechanism passed the functional test. In the performance test, the average consumption of triggering one inter-core interrupt is about 74 cycles, which is about 0.093 microseconds, and the average time of inter-core interrupt processing is about 3157 cycles, which is about 3.986 microseconds. The communication time of the system to perform multi-core IPC is almost the same compared to single-core IPC. The entire inter-core communication mechanism achieves very low latency with little performance consumption, maintaining the high performance of the entire system.

V. CONCLUSION

In this paper, we propose an inter-core communication method based on signal transmission and shared memory, which has the advantages of flexibility and the ability to transfer various types of data. We have implemented and experimented with it on our own microkernel operating system named Mginkgo. The test results show that it takes about 74 cycles on average to trigger an inter-core interrupt, which is about 0.093 microseconds. The average inter-core interrupt processing time is about 3157 cycles, which is about 3.986 microseconds, and the communication time of the system for multi-core IPC is about 18us, which is the same as that of single-core IPC. The inter-core communication method proposed in this paper achieves very low latency with almost no performance consumption and maintains the high performance of the whole system. The analysis of the test results shows that the inter-core communication mechanism extended and implemented for mginkgo microkernel can adapt to the multi-core environment and full use of the performance of multi-core processors.

ACKNOWLEDGMENT

This work is supported by the Introducing Program of Dongguan for Leading Talents in Innovation and Entrepreneur (Dongren Han [2018], No. 738).The corresponding author is Lei Luo.

REFERENCES

[1] Moore, Gordon E. "Cramming more components onto integrated circuits." Proceedings of the IEEE 86.1 (1998): 82-85.
[2] Zheng, Si-Qing, and Jie Wu. "Dual of a complete graph as an interconnection network." Journal of Parallel and Distributed Computing 60.8 (2000): 1028-1046.
[3] Reichenbach, Marc, Benjamin Pfundt, and Dietmar Fey. "Designing and manufacturing of real embedded multi-core CPUs: a holistic teaching approach in computer architecture." 10th European Workshop on Microelectronics Education (EWME). IEEE, 2014.
[4] Pan, Wei, et al. "The new hardware development trend and the challenges in data management and analysis." Data Science and Engineering 3.3 (2018): 263-276.
[5] Roch, Benjamin. "Monolithic kernel vs. Microkernel." TU Wien 1 (2004).
[6] Heiser, Gernot. "Secure embedded systems need microkernels." USENIX; login 30.6 (2005): 9-13.
[7] Munir, Arslan, Ann Gordon-Ross, and Sanjay Ranka. "Multi-core embedded wireless sensor networks: Architecture and applications." IEEE Transactions on Parallel and Distributed Systems 25.6 (2013): 1553-1562.
[8] Nie, Yang, Lili Jing, and Pengyu Zhao. "Design and Implementation of Inter-core Communication of Embedded Multiprocessor based on Shared memory." International Journal of Security and Its Applications 10.12 (2016): 21-30.
[9] Hung, Shih-Hao, Wen-Long Yang, and Chia-Heng Tu. "Designing and implementing a portable, efficient inter-core communication scheme for embedded multicore platforms." 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE, 2010.
[10] Mitropoulou, Konstantina, et al. "Lynx: Using os and hardware support for fast fine-grained inter-core communication." Proceedings of the 2016 International Conference on Supercomputing. 2016.
[11] Wentzlaff, David, et al. "An operating system for multicore and clouds: Mechanisms and implementation." Proceedings of the 1st ACM symposium on Cloud computing. 2010.
[12] Matarneh, Rami. "Multi microkernel operating systems for multi-core processors." Journal of Computer Science 5.7 (2009): 493.
[13] David, Alan. "Scheduling algorithms for asymmetric multi-core processors." arXiv preprint arXiv:1702.04028 (2017).
[14] Balakrishnan, Saisanthosh, et al. "The impact of performance asymmetry in emerging multicore architectures." 32nd International Symposium on Computer Architecture (ISCA'05). IEEE, 2005.
[15] Zhang, Wei. Design and Implementation of Multi-core Support for an Embedded Real-time Operating System for Space Applications. Diss. KTH Royal Institute of Technology, 2015.
[16] Wei, Hongxing, et al. "RT-ROS: A real-time ROS architecture on multi-core processors." Future Generation Computer Systems 56 (2016): 171-178.
[17] Cederman, Daniel, et al. "RTEMS SMP and MTAPI for Efficient Multi-Core Space Applications on LEON3/LEON4 Processors." DASIA 2015-DAta Systems in Aerospace 732 (2015): 24.
[18] Du Sidan, Xing Xianglei Zhou Yu. "On Inter-Processor Communication Mechanism Based on ARM11 MPCore." Computer Applications and Software 5 (2009).
[19] Joshi, Adhiraj, et al. "Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system." Proceedings of the Linux Symposium. 2010.

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.

[20] Gong, Yujian, et al. ”Research on the Technology of Inter-Core Real-Time Communication for Hybrid Multi-System.” 2020 5th Asia Conference on Power and Electrical Engineering (ACPEE). IEEE, 2020.
[21] Chisholm, Micaiah, et al. ”Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems.” 2016 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2016.
[22] Tabish, Rohan, et al. ”An analyzable inter-core communication framework for high-performance multicore embedded systems.” Journal of Systems Architecture (2021): 102178.
[23] Yu, Zhiyi, et al. ”An 800MHz 320mW 16-core processor with message-passing and shared-memory inter-core communication mechanisms.” 2012 IEEE International Solid-State Circuits Conference. IEEE, 2012.
[24] Yu, Zhiyi, et al. ”A 16-core processor with shared-memory and message-passing communications.” IEEE Transactions on Circuits and Systems I: Regular Papers 61.4 (2013): 1081-1094.
[25] Xu, Xiaojie, Network Center, and Lisheng Wang. ”Task assignments based on shared memory multi-core communication.” The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014). IEEE, 2014.
[26] ARM. ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification. England: ARM Limited, 2013, 2-22.

Authorized licensed use limited to: National University Fast. Downloaded on December 01,2022 at 17:06:04 UTC from IEEE Xplore. Restrictions apply.