IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 10 August 2022, accepted 26 August 2022, date of publication 1 September 2022, date of current version 12 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3203702

RESEARCH ARTICLE

Software-Level Memory Regulation to Reduce Execution Time Variation on Multicore Real-Time Systems

SIHYEONG PARK[1], JEMIN LEE[2], AND HYUNGSHIN KIM[3], (Member, IEEE)

[1]SoC Platform Research Center, Korea Electronics Technology Institute, Seongnam-si, Gyeonggi-do 13509, Republic of Korea
[2]Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea
[3]The Division of Computer Convergence, Chungnam National University, Daejeon 34134, Republic of Korea

Corresponding author: Hyungshin Kim (hyungshin@cnu.ac.kr)

ABSTRACT Modern real-time embedded systems are equipped with multi-core processors to execute computationally intensive tasks. In multi-core architecture, last-level cache memory is shared by cores. The shared cache becomes a non-deterministic resource, which affects the independent execution of real-time tasks. We propose a solution to remedy a variation in execution time when interference happens in a shared cache. Current solutions have relied on memory scheduling approaches that avoid concurrent memory access to guarantee deterministic execution time. However, these methods required complex analysis to accurately estimate the worst-case execution time and to schedule tasks in an overly conservative manner. Unlike existing works, the proposed method prevents simultaneous memory access using the side effect of memory barriers rather than the complicated analysis. A memory barrier is inserted based on a simple code analysis that is performed in units of basic blocks using the LLVM compiler. The proposed method not only does not require the modification of the operating system or task execution flow but also relatively shows fast analysis time. To verify the proposed method, we compared the standard deviation of the execution time of each core in a situation where shared cache interference occurs in multi-core. Experimental results show that the proposed basic block-based memory barrier insertion method can reduce the variation in execution time by up to 80% when interference occurs.

INDEX TERMS Code analysis, embedded systems, multi-core architecture, real-time systems, shared cache interference, software-level memory regulation.

I. INTRODUCTION

Multi-core architectures to real-time embedded systems have attracted a lot of attention and has been illustrated to bring tangible computational benefits in many applications over the past few years [1], [2], [3]. However, typical commercial off-the-shelf multi-core platforms have a structure in which cores share a memory hierarchy such as cache, interconnect, and external memory, and as such, memory interference between cores may occur [4]. Memory interference poses a challenge for timing analysis, an essential part of real-time systems. In other words, in a real-time multi-core system, it is crucial to ensure the execution of a task, but it is also important to consider the time constraint of the task [5]. For the time constraint of the system, the execution time should be tightly bounded by minimizing the task execution time variation.

A cache is one of the essential hardware parts because it could improve system performance. The cache can reduce computation time but can also cause interference between tasks in multi-core architectures (e.g., cache miss because of shared cache cleaning for the private cache replacement of other cores) [6]. When a cache miss occurs, data is relocated from the main memory based on the memory hierarchy. At this time, because the bus between the processor and main memory is shared by several platform hardware, performance degradation may occur because of bus contention [7]. This performance degradation depends on the cache replacement policy and cache modeling [8]. An ARM processor for a

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

widely used embedded system takes theoretically one to two and eight cycles to access on-chip L1 and L2 cache, respectively. In main memory case (CoreLink Level 2 Cache Controller L2C-310), an ARM processor takes 30-100 cycles to access main memory [9]. However, as mentioned earlier, if the memory relocation operation is pending because of bus contention, the required cycles to access the main memory become non-deterministic. Therefore, if a cache miss happens, a deadline miss of the task may occur; thus, the interference by the shared cache should be reduced.

The predictable execution model (PREM) has been proposed to remedy interference at the task level by reducing the inter-core memory access in a shared cache [10], [11], [12], [13]. Specifically, PREM isolates each core (or task)'s memory access and reduces cache misses by accessing memory below the private cache size. However, for PREM, it may be necessary to modify the source code or logic of the task and operating system (OS) as well as add a PREM memory access control module. In addition, it takes a long time to analyze because integer linear programming (ILP) analysis is performed for accurate memory access isolation.

We propose a PREM-like method that does not require hardware, task source code, and OS modification in ARM multi-core architecture. By employing a memory barrier [14], the variation in execution time because of shared cache interference is reduced through the prevention of simultaneous memory access of each core in the multi-core. Although memory barriers are typically used to manipulate the order of memory accesses, we utilize the side effect of ARM architecture nature that makes other barriers pending status when a barrier is issued. By doing that, *load* and *store* instructions after a barrier could be blocked. In particular, we propose a method that analyzes the task source code in units of basic blocks with LLVM for counting the amount of memory usage and then inserts a memory barrier based on three thresholds of memory usage: 32, 64, and 128 bytes. Unlike the existing PREM, we are aware of that the proposed method does not fully prevent interference in the shared cache because memory accesses are not thoroughly isolated in each region. Nevertheless, experimental results show that the memory barrier-based PREM-like method can reduce the variation in task execution time by an average of 50% in the shared cache architecture. The contributions to this study are as follows:

- To reduce inter-core interference because of shared cache in multi-core architectures, this study presents an approach to reducing interference through the operational side effect of memory barriers for memory access ordering. This study analyzes the variation in task execution time because of interference, showing that the variation in execution time could be reduced by applying the proposed method.
- The proposed method has the advantage of not requiring any modifications to the execution flow of the task or OS module. In addition, the proposed method can analyze the source code within seconds on the target system.

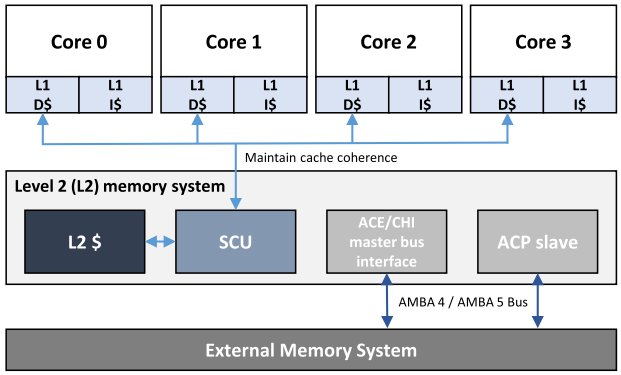


FIGURE 1. Memory hierarchy in ARM Cortex-A53 processor.

- This study presents a basic block-based fine-grained analysis method and passes implementation using LLVM to insert a memory barrier in the task code. In particular, it proposes inserting a memory barrier based on the memory footprint of the basic block.

The structure of this paper is as follows. Section II describes the memory structure and memory barriers of the ARM architecture. Section III deals with the procedure and implementation of the basic block-based memory barrier insertion proposed in this study. Section IV describes the experimental environment and analyzes the interference reduction effect through the proposed method. Section V describes related works, and Section VI describes the limitations of the proposed method and future works. Finally, Section VII provides the conclusion.

II. BACKGROUNDS

This study presents a PREM-like method for reducing shared cache interference using a barrier insertion technique that controls task scheduling. The task is analyzed in units of basic blocks and interference is reduced as a result of the insertion of memory barriers based on the memory footprint of the basic block. In particular, this paper prevents concurrent access to memory by exploiting the operational side effect of memory barriers for memory access ordering. In this section, the memory hierarchy of the ARM Cortex-A architecture, which is the experimental environment in this study, is briefly described, and the operation of the memory barrier, based on this, is explained.

A. MEMORY HIERARCHY OF ARM ARCHITECTURE

Figure 1 shows the memory hierarchy of the ARM Cortex-A53 processor used in this study. We focus on the data cache because it only considers memory access.

Cortex-A53 is equipped with a quad-core processor wherein each core has a data and instruction cache (level 1 (L1) cache). The L1 data cache consists of a physically indexed and tagged cache and operates with a write-back (WB) policy. WB marks a cache line as dirty after a cache write, and updates external memory only when the cache line is evicted or explicitly cleaned.

The level 2 (L2) cache is organized in the L2 memory system as the data cache, and instruction cache is unified. Data in the L2 cache is not filled when it is first fetched from the system, and data allocation occurs only when evicted from the L1 cache. The L2 memory system is composed of the integrated snoop control unit (SCU), advanced microcontroller bus architecture (AMBA) 4 AXI coherency extensions (ACE), AMBA 5 coherent hub interface (CHI) master bus interface, accelerator coherency port (ACP), and an L2 cache. The SCU maintains the coherency of the L1 data cache of each core and connects the four cores to the cluster. The SCU also manages interconnection actions such as arbitration, communication, cache to cache, and system memory transfers [15]. Coherence between caches is performed based on the MOESI state [16]. Data cache uses a pseudo-random cache replacement policy. This policy randomly replaces the next cache line in the set to be replaced, and the victim counter is also selected in a pseudo-random manner. The ACE protocol provides a framework for system-level coherence, maintains correctness in data sharing between caches, and enables interaction between components with different characteristics, maximizing the reusability of cached data. ACP is an AMBA 4 AXI-compatible slave interface that provides core and interconnects points to support read and write transactions without additional coherence requirements. The L2 cache and external memory system are connected by the ACE or CHI bus.

In a symmetric multi-processing (SMP) system (or multi-core processor), in addition to hardware that maintains data consistency between caches, cache maintenance activities performed by code running on a core must be broadcast to other parts of the system [15].

B. MEMORY BARRIER IN ARM ARCHITECTURE

A barrier is a function (instruction) that prevents the re-ordering of memory access instructions, such as out-of-order execution for optimizing system performance. Depending on the processor architecture, the barrier is also called a fence. Data barriers are generally used for data exchange between threads, similar to semaphores, to ensure the order of reading and writing data and for memory-based communication. The ARM architecture supports the following barriers to ordering memory access [17]:

- Data memory barrier (DMB): DMB prevents re-ordering data access instructions through barrier instructions. It ensures that data access operations such as load and store before the barrier instruction are visible to the masters of the corresponding shareable domain, and that the instructions and execution order of all cores after the DMB are guaranteed. It does not affect the order of other instructions executed in core or fetch instructions.
- Data synchronization barrier (DSB): DSB is similar to DMB but has the effect of blocking data access and execution of other instructions until synchronization is complete. However, this does not affect the prefetching of instructions

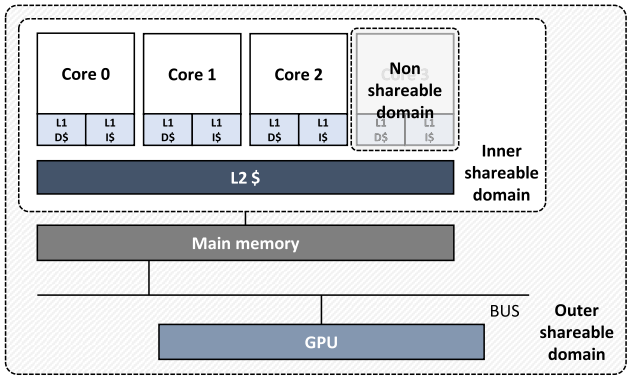


FIGURE 2. Shareable domain in ARM architecture.

In the use of a barrier, the access field parameter can be used as follows [18]:

- Load-load/store: While all loads have to complete before the barrier, stores do not. Loads and stores that appear after the barrier in the program sequence must wait until the barrier is completed.
- Store-store: The barrier only affects the store, and loads can be freely re-ordered.
- Any-any: Both the load and store must be completed before the barrier, and loads and stores after the barrier in program order must wait until the barrier is completed.

Since a wait occurs in processing the memory barrier, the execution time overhead according to the insertion of the barrier must be considered. In addition, since the overhead of the barrier depends on the processor and memory structure, the characteristics of the architecture should also be analyzed. A study analyzing overhead because of memory barriers in ARM architecture showed that throughput could be reduced by up to five times in DSB [19].

In the ARM architecture, the system's memory map is divided into several areas with various access rights, memory types, and cache policies. Therefore, the range of the memory area can be specified when calling the barrier [17]. The memory area is depicted in Figure 2:

- Non-shareable (NSH) domain: An NSH is a memory area only a single process can access. Therefore, an NSH cannot be accessed from other processes. It generally corresponds to the private cache of each processor.
- Inner-shareable (ISH) domain: The ISH is a memory area shared between multiple processors. However, it is not shared with other system domains (e.g., a graphical processing unit (GPU)). Therefore, each processor's private and shared caches are included.
- Outer-shareable (OSH) domain: An OSH is a memory area shared by one or more domains in the system. As OSH includes ISH, it contains the main memory, which can be shared by domains outside the processor, such as a multi-core processor or GPU.
- Full system (SY) domain: The barrier instruction affects all domains by applying a barrier to all system areas.

In the ARM architecture, in the division of shareable domains, hardware modules, such as cores, are expressed as a manager (or master). When the manager issues a barrier, it ensures that other managers in the domain can observe that the transaction has been issued and must be able to observe all transaction issues before the barrier. A barrier transaction is divided into a read barrier transaction that reads the address channel and a write barrier transaction that writes the address channel. At this time, each transaction returns read and write responses for the channel. Furthermore, the interconnect between domains blocks all transactions received after the barrier transaction and issues transactions downstream. The block is removed for downstream issued barrier transactions when it is received into the read and write response channels. Therefore, transactions after the barrier transaction wait until the corresponding barrier transaction response is received [20]. Barrier transaction is provided by the ACE protocol depicted in Figure 1.

This study applies a memory barrier to reduce the interference caused by shared cache between cores. In particular, as this study only considers the memory access part, DMB is used. In addition, because the focus is on the interference between multi-cores, the ISH domain is targeted. As described above, when the barrier is issued by the core (manager), it is controlled by the ACE and ACP interfaces of the L2 memory system. If multiple memory barriers are issued, other barriers are blocked from processing the response to the barrier transaction in a specific domain and interconnect. In addition to this, read and write operations of the corresponding domain are waiting. This study focuses on this barrier transaction process and inserts a barrier in the part that accesses the memory in each core. A side effect that is suspended when memory barriers are issued simultaneously is applied. In other words, the side effect is the delay that occurs because of the response processing of the barrier transaction and the time for which the manager's memory access operation must wait until the end of the barrier. The authors expect it would be possible to reduce the variation in execution time caused by share cache interference by reducing the simultaneous memory access in each core. In addition, this study presents the optimal memory barrier insertion method by analyzing the insertion location and overhead.

III. FINE-GRAIN MEMORY ACCESS CONTROL

This study presents a PREM-like method for reducing shared cache interference using a memory barrier insertion technique that controls simultaneous memory access. This study aims to reduce shared cache interference in real-time multicore systems, where it is essential to bind the task execution time tightly. The source code of the task in units of basic block is analyzed to insert a memory barrier. As a basic block is the smallest unit in the control flow of a system, the proposed approach can achieve better fine-grain memory access control compared with task-level access scheduling. As described in Section II, blocks for additional requested memory barriers occur in processing memory barriers in

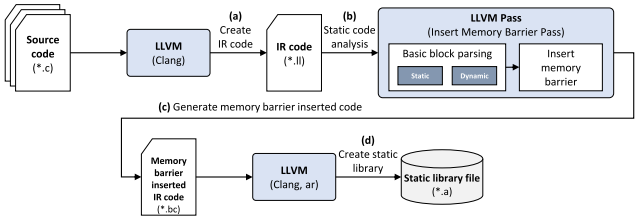


FIGURE 3. Basic block-based memory barrier insertion process.

ACE and interconnect. Moreover, using DMB, the memory operation that occurs after the barrier is pending. Besides, execution time or throughput overhead occurs because of the insertion of memory barriers. To solve this problem, this study also analyzes the overhead caused by the proposed method by subdividing the threshold of the memory footprint of the basic block as a criterion for inserting the memory barrier. The aim here is to minimize concurrent memory access on multiple cores by exploiting these side effects. This section describes the basic block-based memory barrier insertion method proposed in this study.

A. OVERVIEW OF MEMORY BARRIER INSERTION

The overall flow for inserting the memory barrier is shown in Figure 3. The compiler generally divides the program into basic blocks as the first step. A basic block is a straight-line code sequence without branches, apart from entry and exit. Therefore, the code is divided into basic blocks based on branching (*br*) instructions between the blocks and the end of the program (*ret*).

In this study, LLVM was used to convert C code to intermediate representation (IR) code for basic blocks analysis (Figure 3 (a) and (b)), and LLVM Pass was employed to insert barriers (Figure 3 (c)). LLVM is capable of source- and target-independent code generation. After converting the source code, the IR basic block was analyzed using the implemented LLVM Pass. To analyze the basic block, the analysis and insertion steps were separated using two passes: a *Basic Block Parsing Pass* and an *Insert Memory Barrier Pass*. Then, based on the analysis results, an IR code with a memory barrier inserted was generated. This code was compiled as a static library with Clang (version 9.0.0 armv7l) (Figure 3 (d)). An accurate analysis of the basic block is impossible if the extra code for execution time measurement is inserted into the source code. Therefore, this study built a static library and called it from an external code to measure the execution time. C code was converted into an IR using LLVM Clang, and another code was developed to analyze the basic IR blocks using *ModulePass*. For the experiment, bitcode was converted into human-readable IR code (*.ll) using the llvm-dis tool. The *dyn_cast<>* template was used to examine the instructions of each basic block. All basic blocks of the program were statically analyzed for barrier insertion using LLVM Pass. For barrier insertion, the memory footprint of each basic block, which was calculated by examining each barrier's memory access instruction, was used.

In the ARM architecture, load and store instructions control memory access. This study uses the memory alignment values of the instructions to discover the memory footprint of the basic block. The structures of the load and store instructions in IR code are as follows:

```
load i32, i32* %4, align 4
store i32 %69, i32* 5, align 4
```

The alignment of *load* and *store* instructions is considered as the memory footprint of each basic block. Alignment means the size of memory to be accessed in byte size. The *load* instruction requires a pointer operand to read and store a value and the *store* instruction requires the value type and a pointer to store. Therefore, based on this, a specific threshold value is compared to determine whether to insert a memory barrier. In this study, the threshold for memory barrier insertion was set to 32, 64, and 128 bytes based on the L2 data cache line size of Cortex-A53 (64 bytes).

LLVM IR provides fence instruction for memory barriers. The fence instruction is converted into the *dmb ish* ARM instruction, after which it sets the data barrier. *IRBuilder* is used to insert the instruction into the basic blocks with LLVM Pass. *IRBuilder* is an application programming interface (API) that can generate and insert instructions into a basic block's end or a specific part. We use *Builder.CreateFence()* to insert a barrier instruction before the specified instruction of the basic block. The *SetInsertPoint* function is used to insert a barrier next to a specific instruction. If the insertion position is changed to the next instruction (*I.getNextNode()*) using *SetInsertPoint*, the fence is inserted after the current instruction. In LLVM, the *PHINode* or *phi* instruction should always be inserted at the top of the basic block. Therefore, if the first instruction of the basic block is a *phi* node, a barrier instruction is inserted after the corresponding instruction. This study inserts memory barriers at the beginning and end of the basic block; a basic block that accesses a memory size larger than the threshold may cause shared cache interference. As described above, using the memory barrier makes it possible to prevent concurrent memory operation execution because of the delay in the simultaneous barrier processing. In addition, all memory operation execution before the execution of the basic block is completed because of the memory barrier. Therefore, the possibility of interference that may occur with other memory accesses can be reduced for memory operations within the basic block; thus, the variability in execution time may be reduced.

B. BASIC BLOCK PARSING PASS

A *Basic Block Parsing Pass* is a pass that analyzes the code converted to IR, and which is divided into the static and dynamic analyses, as shown in Figure 3.

The first part of the *Basic Block Parsing Pass* statically analyzes the IR. The pass gets the number of all basic blocks of the IR code and the name (number) of the basic block.

```
110:
%111 = load double,double* %6, align 8
%112 = call double@atan(double 1.000000e+00)↩
    #4
%113 = fmul double 4.000000e+00, %112
%114 = fmul double %111, %113
%115 = fdiv double %114, 1.800000e+02
br label %116
```

FIGURE 4. Example of basic block in IR code.

Algorithm 1 Parsing Basic Block in IR Code

```
BasicBlock[] ← 0          ▷ Array for barrier insert classification
MemorySize ← 0            ▷ Memory alignment size of instruction
Index ← 0                 ▷ Basic block number
foreach Function F ∈ Module do
    foreach BasicBlock B ∈ F do
        foreach Instruction I ∈ B do
            if I = StoreInst then
                MemorySize ← MemorySize + Align
            else if I = LoadInst then
                MemorySize ← MemorySize + Align
            else if I = CallInst then
                if MemorySize ≥ Threshold then
                    BasicBlock[Index] ← True
                else
                    BasicBlock[Index] ← False
                end if
                MemorySize ← 0
                Index ← Index + 1
            end if
        end for
        if MemorySize ≥ Threshold then
            BasicBlock[Index] ← True
        else
            BasicBlock[Index] ← False
        end if
        MemorySize ← 0
        Index ← Index + 1
    end for
end for
```

When the example IR code in Figure 4 is executed, it moves to the *@atan* basic block with a call instruction. When the execution of the corresponding basic block is finished, it returns to basic block *#110*. To identify the precise memory footprint of basic block *#110*, it should be divided into two parts based on the call instruction. Algorithm 1 provides a pseudo-code that analyzes the memory footprint of the basic block with consideration of this call flow.

The number of all basic blocks is allocated to the *BasicBlock[]* array. The number of all basic blocks is allocated to the *BasicBlock[]* array, used to determine whether to insert a barrier into the corresponding basic block. This study uses the Index variable to indicate the location of the barrier. The basic block memory footprint (*MemorySize*) is used as the barrier insertion criterion, the sum of the alignment values of the basic block *load* and *store* instructions. The *Basic Block Parsing Pass* sequentially searches all modules, functions, basic blocks, and instructions using a *"for"* statement. If the instruction is a *load* or *store* while searching for basic block

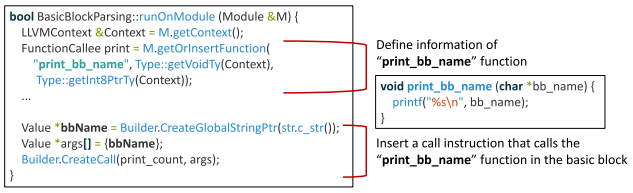


FIGURE 5. Prints the name of the basic block example code.

Algorithm 2 Insert a Memory Barrier Into the Basic Block in IR Code

```
1: BasicBlock[]                                  ▷ Array for barrier insert classification
2: Index ← 0                                     ▷ Basic block number
3: foreach Function F ∈ Module do
4:     foreach BasicBlock B ∈ F do
5:         foreach Instruction I ∈ B do
6:             if BasicBlock[Index] = True then
7:                 if I = First Instruction then
8:                     if I = PHI Node then
9:                         InsertBarrierToNextInstruction(Acquire)
10:                    else
11:                        InsertBarrier(Acquire)
12:                    end if
13:                end if
14:                if I = BranchInst then
15:                    InsertBarrier(Release)
16:                else if I = ReturnInst then
17:                    InsertBarrier(Release)
18:                end if
19:                if I = CallInst then
20:                    InsertBarrier(Release)
21:                    Index ← Index + 1
22:                end if
23:            end if
24:        end for
25:        Index ← Index + 1
26:    end for
27: end for
```

#110, as shown in Figure 4, the align value is added to the *MemorySize* variable (lines 8–19 of Algorithm 1). If the instruction is a call instruction (third line in Figure 4), and the *MemorySize* of the basic block is greater than the threshold value, the corresponding index value of *BasicBlock[]* is set to *"true."* Then, the Index is increased by one and verified with the following instruction (fourth line in Figure 4).

When the search for all instructions in the basic block is finished, the memory size of the current *Index* and the *Threshold* is compared. Then, the decision must be made on whether or not to insert the barrier and write to *BasicBlock[]*. If *MemorySize* is less than Threshold, *"false"* is stored in *BasicBlock[Index]*. Then, for the following basic block check, *MemorySize* is set to zero, increasing the *Index*.

The second part of the *Basic Block Parsing Pass* process involves inserting an output statement to print each basic block's name at runtime (Figure 3: *Dynamic* in *Basic Block Parsing Pass*). This process involves code analysis and is not included in the actual code execution for performance measurement. It is necessary to identify how many basic blocks are executed; the basic block with a barrier inserted is executed to compute the barrier insertion overhead. This study implements a function that receives the basic block name as an argument and prints it, as shown in Figure 5. The *IRBuilder getOrInsertFunction()* is used to pass the number of each basic block as an argument to the function. The *IRBuilder CreateCall()* inserts a call instruction that invokes the output function in each basic block of the IR code. By doing this, all the basic blocks executed at runtime are identified. The execution rate of the basic block can then be calculated.

C. MEMORY BARRIER INSERTION PASS

BasicBlock[] is a check value for memory barrier insertion for all basic blocks, which is derived from the result of the previous *Basic Block Parsing Pass*. Accordingly, the *Insert Memory Barrier Pass* inserts a memory barrier into the entry and exit of the corresponding basic block while searching for the basic block of the IR code. Algorithm 2 presents the pseudo-code of the *Insert Memory Barrier Pass*.

Like the *Basic Block Parsing Pass*, the *Insert Memory Barrier Pass* searches the module, basic block, and instruction. If *BasicBlock[Index]* is *"true,"* the *fence* instruction is inserted at the beginning and end of the basic block. The *Fence Release* instruction is inserted before the respective *branch*, *return*, and *call* instructions (lines 15–22 of Algorithm 2). If the basic block is a *PHINode*, a *Fence Acquire* instruction is inserted after the phi instruction. Then, the index value increases, and *BasicBlock[Index]* verification is repeated with all basic blocks.

IV. EXPERIMENTS

To demonstrate the efficacy of the proposed method, experiments were performed on a Raspberry Pi 3 Model B (RPi3) equipped with a Broadcom BCM2837 quad-core chipset based on ARM Cortex-A53. RPi3 has 32 KB of L1 instruction cache and data cache per core. The L2 cache is 512 MB in size and is shared by four cores. It consists of a unified instruction and data cache, and the cache line is 64 bytes. RPi3 has a main memory of 1 GB. Debian-based Raspbian 32-bit was used as the OS. For the experiment, the PREEMPT_RT patch [21][1] was applied to improve the real-time properties of the Raspbian default kernel (kernel version 5.4.51).

For experimental measurements in PRi3, the performance monitor unit (PMU) of Cortex-A53 was used. The target board provides one cycle counter and six event-based performance counters. this study used *L1D_CACHE_REFILL*, *L1D_CACHE*, *L2D_CACHE*, and *L2D_CACHE_REFILL* event counters. Because the PMU of Cortex-A53 does not provide a cache miss counter, the cache miss rate was calculated by dividing the cache access by the cache miss refill. In addition, the cycle counter of Cortex-A53 was used to measure the execution time of the benchmark.

For the experiment, MiBench [22] was used. It consists of 35 benchmarks for embedded applications. Four benchmarks

[1] https://github.com/raspberrypi/linux/tree/rpi-4.19.y-rt

TABLE 1. Description of selected benchmarks.

Benchmark Suite	Benchmark	Description
Automotive	Basicmath (small, large)	Calculate road speed or vector values using the integer square root, random equations, cubic polynomial, and convert between radians and degrees
	Bitcount (small, large)	Calculate the number of bits of an integer array using bit counting, Optimized 1 bit/loop counter, Ratko's mystery algorithm, Recursive bit count by nybbles, Non-recursive bit count by nybbles
	Qsort (small, large)	Sorts the input string array in ascending order with the quick sort algorithm. Qsort_large consists of three-tuples representing data points
Telecomm	FFT (small, large)	Fast Fourier transforms (FFT) are performed on the float data array. The input to the Fourier transforms operation is a polynomial function with pseudo-random amplitude and frequency sinusoidal components

TABLE 2. Execution arguments and the number of basic blocks per benchmark.

Benchmark		Execution argument	Runtime executed number of basic block
Basicmath	small	loop = 1001	347,210
	large	loop = 10000	12,075,561
Bitcount	small	iteration = 75000	19,000,274
	large	iteration = 112500	284,626,253
Qsort	small	10000 integers	511,740
	large	50000 integers and characters	2,504,751
FFT	small	MAXSIZE = 4096, MAXWAVES = 4	413,901
	large	MAXSIZE = 32768, MAXWAVES = 8	4,669,701

from the Automotive and Telecomm suites were selected and each benchmark function consisted of small and large datasets. Parts that could affect execution time variabilities for the experiment, such as file input/output and print statements included in each benchmark code, were removed. Table 1 describes the benchmarks selected for the experiment and Table 2 shows the arguments used for each benchmark and the number of basic blocks for execution. Each benchmark was run independently on the assigned core, and there were no shared variables or interconnections with other benchmarks. All experiments in this study were repeated 200 times.

A. MEMORY BARRIER INSERTION OVERHEAD

As described in Section II, execution time overhead occurs because of memory barrier insertion. Therefore, in this study, based on the memory alignment of each basic block, a memory barrier was inserted according to a specific threshold, and the performance overhead was analyzed accordingly. The execution time (*cycles*) of the benchmark was measured and the performance overhead (increase in execution time)

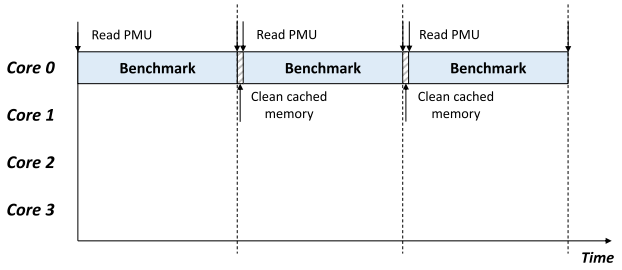
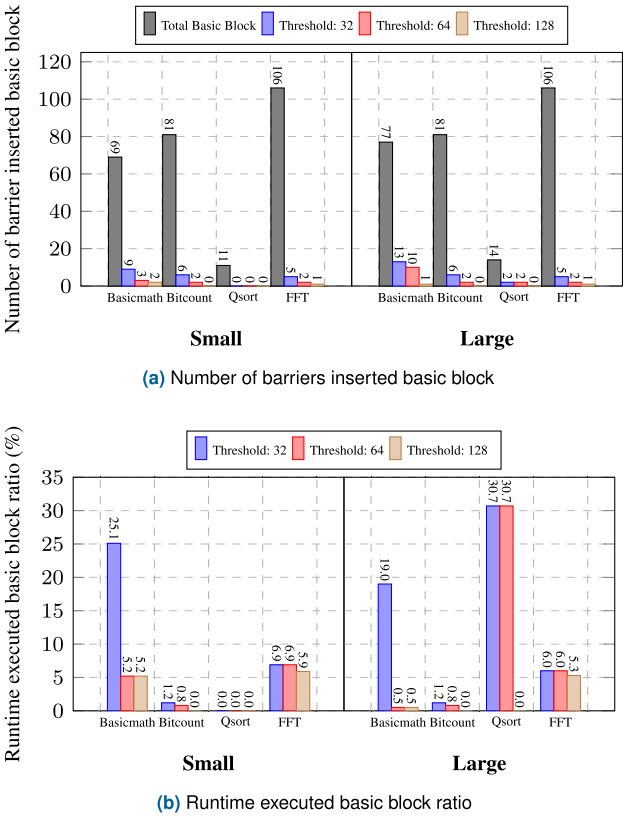


FIGURE 6. Overhead measurement process.

because of the insertion of memory barriers was analyzed. For the experiment, as shown in Figure 6, the benchmark in Table 1 was executed using only *Core 0*, and the cycle counter of Cortex-A53 was read at the start and end points of the benchmark to calculate the cycle elapsed. In addition, cached memory data was cleaned for each benchmark execution.



(a) Number of barriers inserted basic block

(b) Runtime executed basic block ratio

FIGURE 7. Barrier insertion and barrier execution rate by the benchmark.

1) ANALYSIS OF MEMORY BARRIER INSERTION RATIO

Before measuring the overhead of memory barrier insertion, the number of barriers inserted according to the threshold for each benchmark was analyzed. For this purpose, the insertion ratio according to the static and dynamic analysis of the *Basic Block Parsing Pass* of Section III was measured. In addition,

because the basic blocks in the IR code are not all executed according to the execution condition, this study counted the execution rate of the basic block with the memory barrier inserted at runtime. Figure 7 shows the static and dynamic analysis results according to each threshold.

Figure 7 shows the number of basic blocks on the IR code for each benchmark through static analysis and the number of basic blocks with memory barriers inserted according to the threshold. The *y*-axis is the number of basic blocks with memory barriers inserted. Figure 7b shows the ratio of basic blocks inserted with memory barriers for each threshold executed at runtime. The *y*-axis represents the ratio of the basic block in which the memory barrier is inserted in the executed basic block. The *x*-axis of each graph is the benchmark for the dataset.

When the threshold is 32, in the benchmarks of the small dataset, a memory barrier is inserted into the basic blocks of about 13.0, 7.4, 0.0, and 4.7%, respectively. At this time, in each benchmark, the basic block with a memory barrier inserted showed 25.1, 1.2, 0.0, and 6.9% execution proportions at runtime, respectively. In the large dataset, barriers were inserted in 16.9, 7.4, 14.3, and 4.7% of the basic blocks. Execution of the basic block with barriers inserted at runtime showed 19.0, 1.2, 30.7, and 6.0% of the execution, respectively. In the case of a small dataset, Qsort performed the quick sort on 10,000 integers, as shown in Table 2, but no memory barrier was inserted in all 11 basic blocks. This means that the sum of alignment of each basic block did not exceed 32 bytes, and its analysis showed that a large-sized memory operation did not occur according to the repeated access to integer variables. In contrast, Qsort of the large dataset was inserted into 14.3% of the basic blocks and showed a weight of 30.7% in the overall execution. The large dataset received and sorted mixed integer and char data as input, and it can be seen that this was because more variables were allocated and used than the source code of the small dataset. When the threshold was 64, 4.3, 2.5, 0.0, and 1.9% of the basic blocks were inserted in the benchmarks of the small dataset, respectively, and 5.2, 0.8, 0.0, and 6.9% were executed at runtime. The large dataset was inserted into 13.0, 2.5, 14.3, and 1.9% of the basic blocks and occupied the runtime proportions of 0.5, 0.8, 30.7, and 6.0%, respectively. When the threshold was 32, the proportion of insertions and executions in Basicmath and Bitcount decreased in all data sets. When the threshold was 128, basic blocks were inserted at 2.9, 0.0, 0.0, and 0.9% of the small dataset benchmarks, respectively, and were executed at 5.2, 0.0, 0.0, and 5.9% at runtime. In the case of Basicmath and FFT, the proportion of inserted basic blocks was lower than that of the case where the threshold was 64, but the proportions executed at runtime were the same. In the large dataset, the insertion ratios of memory barriers were 1.3, 0.0, 0.0, and 0.9%, respectively, and the execution ratios were 0.5, 0.0, 0.0, and 5.3% at runtime. In the case of Qsort, when the threshold was 64, memory barriers were inserted in 30.7% of basic blocks, but none when the threshold was 128.

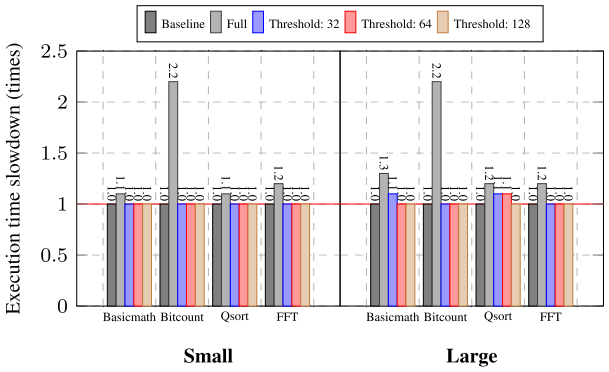


FIGURE 8. Overhead because of barrier insertion.

2) ANALYSIS OF EXECUTION TIME OVERHEAD ACCORDING TO MEMORY BARRIER INSERTION

Figure 8 shows the execution time overhead according to the memory barrier insertion. For the experiment, the *Insert Memory Barrier Pass* of Section III was used. It was also measured when the memory barrier was inserted in all basic blocks for each benchmark for comparison. As a result of the experiment, the execution time of all benchmarks increased when a memory barrier was inserted in all basic blocks. In particular, in the case of Bitcount, the execution time of small and large datasets increased by 2.2 times. Since the Bitcount benchmark uses several algorithms for bit count, there are fewer repetitively used codes than in other benchmarks; further, as shown in Table 2, the total number of basic blocks executed is more significant than in other benchmarks. Therefore, it is concluded that the execution time increases significantly when a memory barrier is inserted in the entire basic block.

In the small dataset, the increase in execution time was insignificant for the insertion of memory barriers according to all thresholds. In the benchmark of the large dataset, when the threshold was 32, Basicmath and Qsort had a 1.1-fold increase in execution. Bitcount and FFT did not affect the execution time, although the basic block with a memory barrier inserted at runtime showed a 1.2 and 6.0% execution proportion, respectively, as shown in Figure 7b. When the threshold was 64, only Qsort increased the execution time by 1.1 times. Even though a memory barrier was inserted in the Basicmath and FFT, there was no change in execution time when the threshold was 128, as in the case of 64.

Through the threshold-based memory barrier insertion method proposed in this study, a memory barrier is inserted except for Qsort of a small dataset. In particular, at thresholds 32 and 64, the execution time of some benchmarks on large datasets increased by 1.1 times, and there was no significant change in others. Therefore, it can be seen that the execution time overhead according to the insertion of the memory barrier is not significant.

B. INTERFERENCE REDUCTION ANALYSIS

This section analyzes the interference caused by the shared cache between cores and explains the effect of the basic

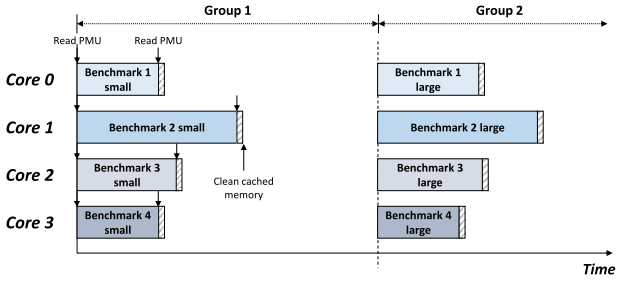
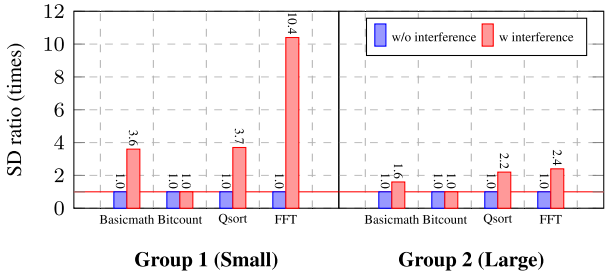
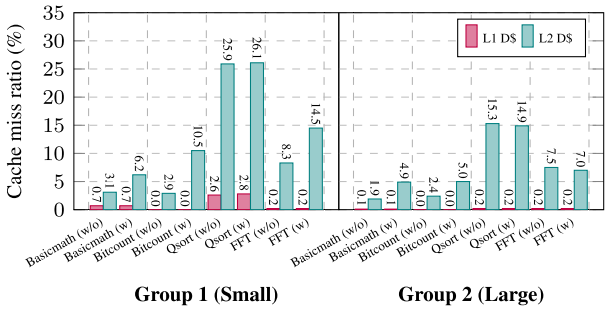


FIGURE 9. Benchmark experiment with multi-core architecture.



(a) Execution time distribution with interference



(b) Cache miss ratio variation with interference (w/o denotes without interference, w denotes with interference)

FIGURE 10. Measurements of impact of interference.

block-based memory barrier insertion proposed in this study. For the experiment, as shown in Figure 9, each benchmark was assigned to each core and executed simultaneously. For this, Linux's *sched_setaffinity()* was used to dedicate benchmarks to each core as much as possible. In addition, the *cycle counter* and *PMU register* were read at the beginning and end of each benchmark execution, and when the execution was over, the cached data was cleaned. For the experiment, we divided benchmarks into two groups according to the size of the dataset: *Group 1* (small dataset) and *Group 2* (large dataset). In each group's experiment, Basicmath, Bitcount, Qsort, and FFT were assigned to *Cores 0, 1, 2,* and *3*.

1) ANALYSIS OF INTERFERENCE BETWEEN CORES

To confirm the effect of the memory barrier insertion, the effect of inter-core interference was analyzed. Figure 10 shows the interference effect when executed simultaneously by the group whereas Figure 10a shows the deviation in execution time because of inter-core interference. The *y*-axis shows the ratio of standard deviations. Figure 10b shows L1, and L2 cache misses in benchmark execution by group; the *y*-axis is the cache miss ratio. The *x*-axis of each graph represents the benchmark of each group. Moreover, the without (w/o) interference bar on the *x*-axis of each graph refers to the value when running each benchmark with a single core.

The benchmark of the small dataset, excluding Bitcount, Basicmath, Qsort, and FFT, showed 3.6, 3.6, and 10.4 times standard deviation changes in execution time, respectively. As the standard deviation of the execution time increases, the tight bounding of the execution time becomes impossible. Although most of the benchmarks did not increase the L1 cache miss ratio, the L2 cache miss increased up to approximately 3.6 times. In the case of Bitcount, while the L2 cache miss ratio increased from 2.9 to 10.5%, the standard deviation of the execution time did not change significantly. In contrast, in Qsort, the L2 cache miss increased by 0.2%, but the standard deviation of the execution time was large. As shown in Table 2, as Bitcount's execution time was the longest, it is concluded that the effect of interference is minor, even if it is run simultaneously with other benchmarks.

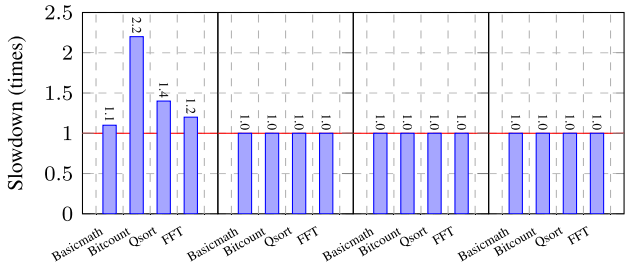
In the case of the large dataset, similar to that of the small dataset, the execution time standard deviations of Basicmath, Qsort, and FFT increased by 1.6, 2.2, and 2.4 times, respectively, except for Bitcount. In the cache miss ratio, Basicmath and Bitcount more than doubled, while Qsort and FFT decreased slightly.

2) ANALYSIS OF VARIABILITY IN EXECUTION TIME BECAUSE OF MEMORY BARRIER INSERTION

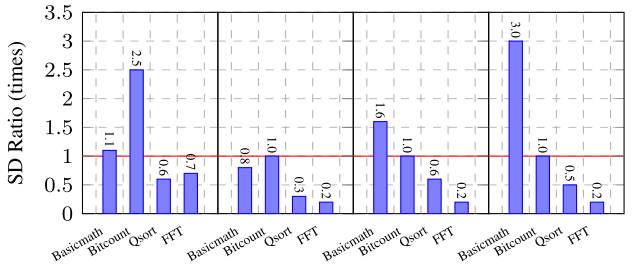
To insert the memory barrier through the basic block analysis proposed in this study, the *Insert Memory Barrier Pass* shown in Figure 3 was used. To analyze the change in execution time because of the insertion of the memory barrier, the experimental method shown in Figure 9 was followed, as in the previous experiment. Moreover, for comparison, the insertion of memory barriers in all basic blocks was measured.

Figure 11 shows the experimental results on a small dataset. The *x*-axis of each graph shows the experimental results according to the threshold, and each benchmark indicates *Core 0-4* (for comparison with the existing graph, the benchmark's name was written instead of the core number for convenience). Figure 11a shows the increase in execution time because of memory barrier insertion and interference, Figure 11b shows the standard deviation ratio of execution time, and Figure 11c shows the amount of change in the cache miss ratio for each case. The value of each graph represents the ratio of the value in case of interference in Figure 10.

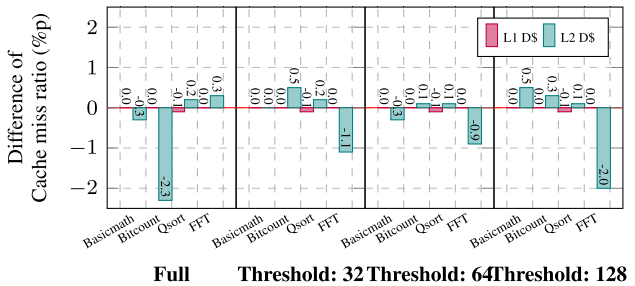
When a memory barrier is inserted in all basic blocks (*full* in Figure 11), the execution time changes by about 1.1 to 2.2 times. The increase in execution time was similar to the overhead caused by the insertion of the memory barrier in Figure 8, but in the case of Qsort, it increased by 0.3 times because of interference. This means that the



(a) Slowdown



(b) SD ratio



(c) Difference of cache miss ratio

FIGURE 11. Barrier insertion result of small dataset benchmarks.

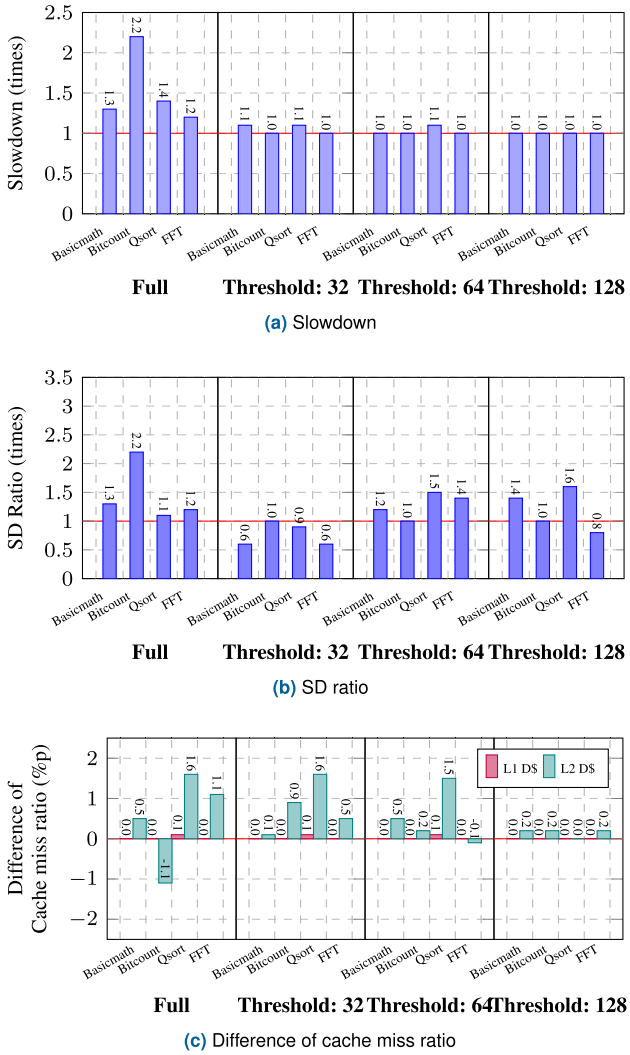
overhead because of the memory barrier is greater than the increase in execution time because of interference. For the standard deviation of execution time, while Basicmath increased by 1.1 times and Bitcount by 2.5 times, Qsort and FFT decreased by 0.6 and 0.7 times, respectively. Each benchmark's change in cache miss ratio showed little change in the L1 cache, but L2 cache miss in Basicmath and Bitcount decreased by 0.3 and 2.3%, respectively. The L2 cache miss ratios of Qsort and FFT increased by 0.2 and 0.3%, respectively. When the threshold was 32, there was hardly any increase in each benchmark's execution time. The standard deviation change of execution time also decreased for all except Bitcount. In particular, in the case of FFT, the standard deviation of the execution time decreased by 0.2 times compared to the case where interference occurred because of the insertion of the memory barrier. In contrast, in the case of cache misses, excluding FFT, L2 cache misses increased by up to 0.5%. Even when the threshold was 64, the execution time of each benchmark did not increase. However, unlike when the threshold was 32, the standard deviation ratio of the execution time increased by 1.6 times in the case of Basicmath. The rest of the benchmarks had standard deviations reduced by 0.2 times. The cache miss ratio was also similar to that of threshold 32. When the threshold was 128, the execution time did not increase, but Basicmath's execution time standard deviation ratio increased by a factor of 3.0. In the case of cache miss ratio, except for FFT, the L2 cache miss ratio slightly increased.

Through experiments on small datasets, the threshold-based memory barrier insertion method did not reduce the execution time compared to the situation where interference occurred but typically reduced the standard deviation of the execution time. In particular, when the threshold was 32, the standard deviation of the benchmark execution time of all cores did not increase in preparation for the interference situation. Moreover, while there was no significant change in the case of a cache miss, it decreased in the execution of some core benchmarks. In the case of Qsort, no memory barrier was inserted for all thresholds, as shown in Figure 7b, but the standard deviation ratio of the execution time was reduced by up to 0.3 times. This means that even if the memory barrier is not inserted, it may be affected by the memory barrier operation performed by other cores.

Figure 12 shows the result of inserting a memory barrier for a large dataset. The axes and expression of the graph are the same as in Figure 10.

When interference occurs, and memory barriers are inserted in all basic blocks, results are similar to that of the memory barrier insertion overhead in Figure 8. Similar to the experimental results of the small dataset, the large dataset also indicates that the overhead caused by the insertion of the memory barrier is larger than the effect of interference when the memory barrier is inserted in all basic blocks. The standard deviation of execution time also increased by up to 2.2 times compared to the interference situation, and cache misses by 1.6%. When the Threshold was 32, the execution times increased by 1.1, 1.0, 1.1, and 1.0 times for each benchmark, but the standard deviations of the execution times were 0.6, 1.0. 0.9 and 0.6 times. In contrast, there was no decrease in the cache miss ratio as it increased by up to 1.6%. When the threshold was 64, the change in execution time did not increase except for Qsort. However, the standard deviation of the execution time and the cache miss ratio also increased up to 1.5 times. When the Threshold was 128, there was no change in the execution time. The standard deviation of the execution time also increased by up to 1.6, excluding the FFT. As shown in Figure 7b, when the threshold was 128, except for the FFT, almost no memory barrier was inserted in the remaining benchmarks, indicating that the standard deviation of the FFT execution time partially improved. Therefore, the cache miss ratio did not show a significant change.

Based on the experimental results of the large dataset, the standard deviation of the execution time was reduced when the threshold was 32, similar to that of the small dataset. However, although the execution time increased by about

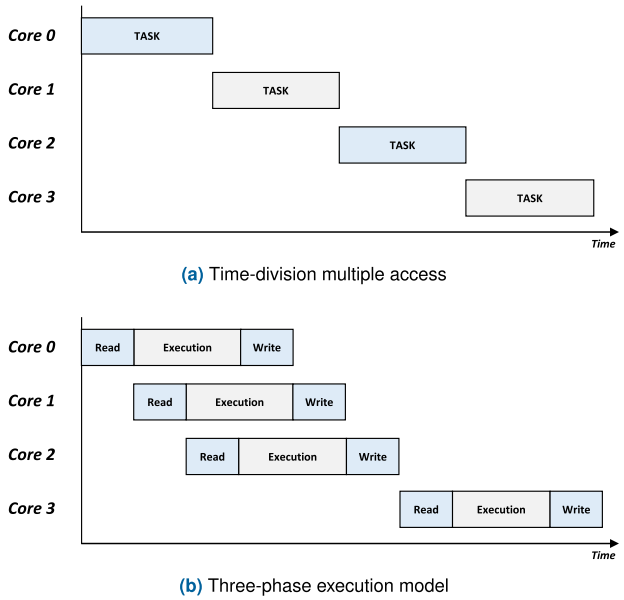


(a) Slowdown

(b) SD ratio

(c) Difference of cache miss ratio

FIGURE 12. Barrier insertion result of large dataset benchmarks.



(a) Time-division multiple access

(b) Three-phase execution model

FIGURE 13. Examples of the execution model.

1.1 times, the change in cache miss ratio because of memory barrier insertion was insignificant.

Through experiments, this study shows that the proposed memory barrier insertion method according to the threshold based on the basic block analysis can reduce the distribution of execution time when interference between cores occurs. In particular, when the threshold was 32, there was no increase in the execution time except in some cases, and the standard deviation of the execution time decreased by up to 0.2 times. The goal here is not to reduce the execution time but to help tightly bound the execution time so that the proposed method is valid. In addition, the proposed method did not have a significant effect on reducing cache misses.

V. RELATED WORKS

The PREM method [10], [11], [12], [13] was proposed to control task scheduling and reduce the interference caused by resource sharing and contention in a multi-core system. PREM arranges the execution of tasks to avoid contention in each core's access to shared resources (e.g., cache memory). Memory-centric scheduling (MCS) [23], [24], [25] was proposed to avoid or limit concurrent access to shared memory.

Task scheduling using time-division multiple access (TDMA) [5], [26] in Figure 13 is a typical MCS approach that executes only one task per globally scheduled time slot. In a multi-core architecture, this approach is inefficient because it allows only one core to run at a time. Therefore, TDMA has low utilization but does not cause inter-core interference. Consequently, the tight bounding of execution time is possible, even in a shared memory structure.

A three-phase execution model was proposed to compensate for the low utilization of TDMA [27], [28], [29], as shown in Figure 13b. The three-phase execution model in Figure 13b is an example of several execution flows. It increases concurrency by dividing the task into a memory-centric (*M*) phase (*"Read"* and *"Write"* in Figure 13b) and a computation (*C*) phase (*"Execution"* in Figure 13b). The M phase prefetch reads data and instructions from the shared global memory to the local memory. During the *C* phase, the processor performs computations with the data. By not accessing shared memory, it avoids contention and can be concurrently executed under the *M* phase. A problem with this model is that either the code must be implemented from scratch, or the legacy code must be modified according to the model.

State-of-the-art [12], [13] three-phase execution models comprise automated code analysis, transformation, and scheduling for PREM execution. These studies aim to avoid contention and eliminate interference between cores. As shown in Table 3, the state-of-the-art PREM methods are compared to the proposed method.

Previous studies performed automated region-based memory profiling for source code transformation using a

TABLE 3. Comparison of state-of-the-art methods with the proposed method.

	State-of-the-art	Proposed method
Goal	Contention-free Region	Contention-allowed Basic block
Optimization level		
Memory footprint analysis	✓	✓
Memory access control	✓ (Under private cache size)	✓ (Based on the memory footprint of a basic block)
Source code refactoring	✓	✗
WCET analysis	✓	✗
Scheduling algorithm	Heuristic [12], Genetic algorithm [13]	✗
Code analysis time	More than several minutes on the desktop [12], [13]	A few seconds on the target device *

* This result was measured in our experimental environment (MiBench on Raspberry Pi 3 model B)

three-phase model. The source code of the task was divided into several segments for this model. Each segment was then configured to be smaller than the core's private cache (e.g., L1) based on the memory footprint used during the code execution. Accordingly, the code was analyzed and loop unrolling and tiling were performed. Each segment consisted of three phases: read, execute, and write. As its memory usage was larger than the original, the transformed code was divided into more segments. Therefore, the time required for memory access isolation increased in other cores.

The worst-case execution time (WCET) was estimated using ILP analysis to optimize the three-phase task scheduling. Based on the results, the execution time of each segment phase was set. Furthermore, a schedule that arranges segments using a genetic algorithm [13] and heuristic method [12] was implemented to ensure optimal execution time without inter-core interference. No variations in the task execution time were experienced, even during inter-core interference.

In a previous PREM study, a method for registering a memory-access-block system call in the kernel area [29] or a memory mutex [12] was used to control each phase. However, owing to the change from user mode to privileged mode, and depending on the OS, using a system call can incur execution time overhead. Moreover, in the source code, controlling memory access is frequently called, and the execution time increases by up to 2.5 times or more owing to prefetch operations during the read phase. Moreover, only single-entry, single-exit codes that form without recursion are applicable to automatic code conversion. Another problem is that ILP computation requires tens of minutes of computation time. Furthermore, ILP analysis has the disadvantage of recalculation when the instruction set architecture is changed.

VI. DISCUSSION AND FUTURE WORKS

This study proposed a memory barrier insertion method based on basic block analysis to reduce the interference caused by a shared cache that may occur in a multi-core real-time system. A benchmark consisting of tasks for a traditional embedded system was used for the experiment. In particular, four benchmarks were selected and tested in two groups according to the amount of input data.

A limitation of this study is the lack of experiments according to the combination of benchmarks with various workloads. The impact of shared cache interference may vary depending on the performance characteristics of each benchmark. In particular, recently, memory-intensive deep learning operations have been applied to real-time systems [30], [31]. Therefore, future work will analyze the interference effect according to the performance characteristics using various benchmarks. In the experiment, the insertion of the memory barrier is decided based on the threshold. At this time, the same threshold value is applied to the benchmark of each core. It is also necessary to consider performance overhead and interference reduction by applying a threshold based on the characteristics of the benchmark.

Another limitation of this study is that, unlike the previously proposed PREM studies, inter-core interference may still occur even if a memory barrier is used. Furthermore, optimizations such as out-of-order cannot be used because of the insertion of memory barriers. Performance degradation because of this part should also be analyzed.

Finally, the operating characteristics of memory barriers differ depending on each architecture's implementation method. Hence, it is necessary to analyze whether the proposed method can reduce the interference caused by the shared cache outside of the ARM architecture.

VII. CONCLUSION

This study aims to reduce the distribution of task execution time because of the interference caused by the shared cache. This can assist in the tight bounding of execution time, which is one of the important factors in a real-time system. The occurrence of interference caused by shared cache contention in a multi-core architecture was analyzed and a method to reduce task execution time variations by inserting memory barriers into the basic blocks of the source code using LLVM Pass was proposed. This study used side effects such as delay of memory operation execution because of the memory barrier and block when simultaneous memory barrier requests occur, and presented a fine-grain analysis method for dividing a basic block based on its call instructions. The memory footprint of each basic block was used for the memory barrier insertion. Through experiments, the execution time overhead according to the insertion of memory barriers was analyzed to show the distribution of execution time by threshold. In particular, when the threshold was 32-byte, because of the insertion of the memory barrier, no increase in execution time was evident. Additionally, it was shown that the standard deviation of the execution time of all core tasks was reduced by up to 80%. In addition, the proposed method has the advantage of not modifying OS or task execution flow.

REFERENCES

[1] J. Athavale, R. Mariani, and M. Paulitsch, "Flight safety certification implications for complex multi-core processor based avionics systems," in *Proc. IEEE Int. Rel. Phys. Symp. (IRPS)*, Mar. 2019, pp. 1–6.

[2] O. Sander, F. Bapp, L. Dieudonne, T. Sandmann, and J. Becker, "The promised future of multi-core processors in avionics systems," *CEAS Aeronaut. J.*, vol. 8, no. 1, pp. 143–155, Mar. 2017.

[3] S.-C. Lin, Y. Zhang, C. H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 751–766, 2018.

[4] C. Courtaud, J. Sopena, G. Müller, and D. G. Perez, "Improving prediction accuracy of memory interferences for multicore platforms," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2019, pp. 246–259.

[5] A. Andrei, P. Eles, Z. Peng, and J. Rosen, "Predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proc. 21st Int. Conf. VLSI Design (VLSID)*, 2008, pp. 103–110.

[6] A. Nogueira and M. Calha, "Predictability and efficiency in contemporary hard RTOS for multiprocessor systems," in *Proc. IEEE 17th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2011, pp. 3–8.

[7] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 272–282, Jun. 2003.

[8] C. Pan, X. Hu, L. Zhou, Y. Luo, X. Wang, and Z. Wang, "PACE: Penalty aware cache modeling with enhanced AET," in *Proc. 9th Asia–Pacific Workshop Syst.*, Aug. 2018, pp. 1–8.

[9] "Corelink level 2 cache controller L2C-310 technical reference manual," ARM, Cambridge, U.K., Tech. Rep., 2012.

[10] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2011, pp. 269–279.

[11] J. Arora, C. Maia, S. A. Rashid, G. Nelissen, and E. Tovar, "Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling," in *Proc. 29th Int. Conf. Real-Time Netw. Syst.*, Apr. 2021, pp. 123–133.

[12] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu, "Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution," in *Proc. 9th Int. Workshop Program. Models Appl. Multicores Manycores*, New York, NY, USA, Feb. 2018, pp. 11–20.

[13] B. Forsberg, M. Mattheeuws, A. Kurth, A. Marongiu, and L. Benini, "A synergistic approach to predictable compilation and scheduling on commodity multi-cores," in *Proc. 21st ACM SIGPLAN/SIGBED Conf. Lang., Compil., Tools Embedded Syst.*, New York, NY, USA, Jun. 2020, pp. 108–118.

[14] P. E. McKenney, "Memory barriers: A hardware view for software hackers," Linux Technol. Center, IBM Beaverton, Tech. Rep., 2010.

[15] "Arm cortex-A53 MPCORE processor, technical reference manual," ARM, Cambridge, U.K., Tech. Rep., 2014.

[16] T. Suh, D. M. Blough, and H.-H.-S. Lee, "Supporting cache coherence in heterogeneous multiprocessor systems," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, 2004, pp. 1150–1155.

[17] "Arm cortex—A series programmer's guide for ARMv8-A," ARM, Cambridge, U.K., Tech. Rep., 2015.

[18] "ARMv8—A memory systems," ARM, Cambridge, U.K., Tech. Rep., 2016.

[19] N. Liu, B. Zang, and H. Chen, "No barrier in the road: A comprehensive study and optimization of ARM barriers," in *Association for Computing Machinery*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 348–361.

[20] "AMBA AXI and ACE protocol specification," ARM, Cambridge, U.K., Tech. Rep., 2021.

[21] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time Linux kernel: A survey on PREEMPT_RT," *ACM Comput. Surveys (CSUR)*, vol. 52, no. 1, pp. 1–36, Feb. 2019.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization (WWC)*, Dec. 2001, pp. 3–14.

[23] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Syst.*, vol. 48, no. 6, pp. 681–715, Nov. 2012.

[24] J. M. Aceituno, A. Guasque, P. Balbastre, J. Simó, and A. Crespo, "Interference-aware schedulability analysis and task allocation for multicore hard real-time systems," *Electronics*, vol. 11, no. 9, p. 1313, Apr. 2022.

[25] A. Guasque, J. M. Aceituno, P. Balbastre, J. Simó, and A. Crespo, "Schedulability analysis of dynamic priority real-time systems with contention," *J. Supercomput.*, pp. 1–23, Apr. 2022.

[26] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in *Proc. 16th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2010, pp. 215–224.

[27] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *Proc. 21st IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2015, pp. 285–296.

[28] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in *Proc. 23rd Int. Conf. Real Time Netw. Syst.*, New York, NY, USA, Nov. 2015, pp. 87–96.

[29] J. M. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo, "Implementation of memory centric scheduling for COTS multi-core real-time systems," in *Proc. 31st Euromicro Conf. Real-Time Syst. (ECRTS)*, Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019, pp. 1–23.

[30] A. Alharbi, W. Alosaimi, R. Sahal, and H. Saleh, "Real-time system prediction for heart rate using deep learning and stream processing platforms," *Complexity*, vol. 2021, pp. 1–9, Feb. 2021.

[31] Z. Zhao, K. Wang, N. Ling, and G. Xing, "EdgeML: An AutoML framework for real-time deep learning on the edge," in *Proc. Int. Conf. Internet-Things Design Implement.*, New York, NY, USA, May 2021, pp. 133–144.

SIHYEONG PARK received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Chungnam National University, in 2014, 2016, and 2021, respectively. He is currently a Senior Researcher at the Korea Electronics Technology Institute (KETI). His research interests include multi-core embedded systems and real-time systems.

JEMIN LEE received the B.S. and Ph.D. degrees in computer science and engineering from Chungnam National University, in 2011 and 2017, respectively. From 2017 to 2018, he was a Postdoctoral Researcher at the Korea Advanced Institute of Science and Technology (KAIST). He is currently a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI). His research interests include energy-aware mobile computing and deep learning compiler.

HYUNGSHIN KIM (Member, IEEE) received the B.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1990, the M.Sc. degree in satellite communication engineering from the University of Surrey, U.K., in 1990, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2003. In 2004, he joined Chungnam National University and he is currently a Professor with the Department of Computer Science and Engineering. His research interests include real-time embedded software and embedded AI computing.