IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 13, 2019, accepted April 2, 2019, date of publication April 10, 2019, date of current version April 17, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2909642

A Thread-Oriented Memory Resource Management Framework for Mobile Edge Computing

ZONGWEI ZHU[1], FAN WU[1], JING CAO[1], XI LI[1], AND GANGYONG JIA[2]
[1]Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215000, China
[2]Department of Computer Science, Hangzhou Dianzi University, Hangzhou 310000, China

Corresponding author: Fan Wu (wf18@mail.ustc.edu.cn)

ABSTRACT Increasing the number of cores is one of the most effective methods to enhance performance. However, an extensive experimental study on mobile edge computing (e.g., Android devices) indicates that the memory management system has gradually become a key performance bottleneck. Studies on improving memory management mainly focus on exploring the trade-off between avoiding fragmentation and improving allocation efficiency. From our previous research, we know that the fragmentation is no longer a crucial bottleneck; instead, inter- and intra-thread behavior should be focused on, and thus, we introduce memory management based on thread behaviors (MMBTB). Unfortunately, it lacks a unified optimization program interface and good architecture. Consequently, in this paper, we propose a memory resource management at operating system (OS) layer of mobile edge computing, called the thread-oriented memory management layer (TOMML) to address this problem, which follows the microkernel architecture pattern and can meet the user's requirements for selecting plug-ins to achieve different optimization goals. This paper is divided into several sections as follows. First, we demonstrate the efficiency of TOMML through theoretical simulation and experimentation. The experimental result is that TOMML can improve memory allocation efficiency by 12%–20%. Furthermore, we introduce a plug-in to save power, which can further promote 6%–25% bank free compared with previous excellent research.

INDEX TERMS Memory management, power control, parallel algorithms, edge computing, thread behaviors.

I. INTRODUCTION

Driven by the growth of the Internet of Things (IoT) and cloud services, a new computing paradigm is becoming more and more popular, edge computing, in which data processing are placed at the network edge in close proximity to mobile devices [1], [2]. This emerging technology has the potential to solve response time requirements as well as data privacy and safety, but this will place highly demands on the power of resource-limited mobile devices. Therefore, multicore processors are becoming a common choice for manufacturers especially when it is difficult to increase the clock frequency, such as *Freescale*, *MediaTek*, *Samsung*, and *Texas Instruments*. They have introduced dual-core, quad-core, and even eight-core processors to enhance multicore processor performance. Unfortunately, it has been shown that the shared main memory is still a bottleneck of system performance. Therefore, how to optimize the memory management system has become an urgent problem that needs to be solved in industry. Meanwhile, memory management has attracted considerable attention from academia since it immensely affects the whole performance [3]. Modern multicore systems may not be able to directly adopt traditional memory management systems, such as *segregated fits*, *sequential fits*, *best fits*, and *simple segregated storage*. Xiao *et al.* [4] propose a novel methodology to model the dynamic execution of an application and partition the application into an optimal number of clusters for parallel execution. But, the program needs to be analyzed first and divided into clusters for parallel execution. This may not be suitable for individual users. Research [5]–[9] on improving memory management focuses primarily on exploring the trade-off between improving

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

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

distribution efficiency and reducing fragmentation. However, most existing research does not specifically analyze inter- and intra- thread behaviors.

For OS (e.g., Linux, Android) of mobile edge computing devices, external fragmentation has a large impact on overall performance [9], so the global buddy system is used to allocate memory to minimize fragmentation and provide continuous physical pages. However, according to the previous research on thread behavior and lock contentions, we obtain different conclusions. First, in most mobile apps, single-page requests account for more than 96% of total memory requests and external memory fragmentation rarely becomes a bottleneck in system performance. Although the Linux kernel introduces *per-CPU* for single-page memory requests to speed up single-page allocation, the algorithm with *O(logn)* complexity *(n being the size of the memory)* is too high for hardware and is not suitable for mobile edge computing devices [5]. Second, in multicore systems, the lock contentions will dominate the execution time as the number of cores increases.

As described above, the external fragmentation issues may not be as important as previously assumed. However, most previous research only focus on decreasing the algorithm complexity to improve memory allocation efficiency [5]–[9] and ignore the impacts of inter- and intra- thread behaviors. Therefore, in the previous study, we introduced MMBTB to enhance memory management efficiency depending on thread behavior. Unfortunately, due to the traditional architecture adopted by MMBTB [10], it lacks good scalability and sufficient overall agility. This makes MMBTB unable to adapt to the fast-changing Android devices. Similarly, in terms of functions, it cannot meet the multiclass and high-quality demands of users. More specifically, first, MMBTB and the threads are tightly coupled and may not be completely transparent to the threads, which causes unpredictable problems when new threads are forked by mobile system updating. Then, MMBTB is process-oriented, and the entire program flow is designed to enhance the efficiency of memory allocation. This makes it difficult to develop a unified optimization program interface. That is, the users cannot change optimization targets in different scenarios. Consequently, it is important to propose a new memory management system as a solution to address this problem.

In this paper, we introduce a novel memory resource management at OS layer of edge computing devices, called TOMML, which takes the thread as the basic unit for managing memory resources and locates it between the native layer and the framework layer. Compared with MMBTB, our main contributions are the following. (1) TOMML adopts the microkernel architecture, and it is divided between independent plug-in modules and the basic core system. TOMML can meet the user's requirements to select plug-ins to achieve different optimization goals. In addition, TOMML is completely transparent to threads, and in theory, it will not cause any harmful effect on the new thread operation. (2) TOMML internal modules have clear functions, responsibilities and relationships with each other. For example, the thread interface module, plug-in database module, and plug-in interface module have the functions of collecting thread features, configuring plug-in running resources and environment, and implementing different plug-ins. Each programmer can separately develop plug-ins with different functions. In summary, TOMML speeds up the development of plug-ins and realizes collaborative development. Therefore, to prove the scalability of the plug-in interface layer, we make and test a plug-in on the Android platform to save dynamic random access memory's (DRAM's) self-refresh power.

The rest of this paper is organized as follows. The next section discusses the TOMML scheme in detail. Methods for experimental research and analysis of results are presented in Section 3. Section 4 discusses the related work, and finally, Section 5 concludes.

II. PROPOSED MECHANISMS

A. THREAD MEMORY OPERATING BEHAVIOR

Thread behavior can be divided into single-thread behavior and multithread interactions. First, in terms of a single thread, as depicted in Fig. 1 (the *x*-axis represents the thread's life cycle, and the *y*-axis represents its memory resources), it has two operations to manipulate memory resources, *request* and *release* (TL represents a thread's lifetime), and three lifetime stages including *creative, active* and *exit*. As shown in Fig. 1, the demand for memory dramatically increases when the thread is in the creative stage, and all the memory resources are released when the thread exits. In addition, usually the active stage is the longest stage in the lifetime of the thread.

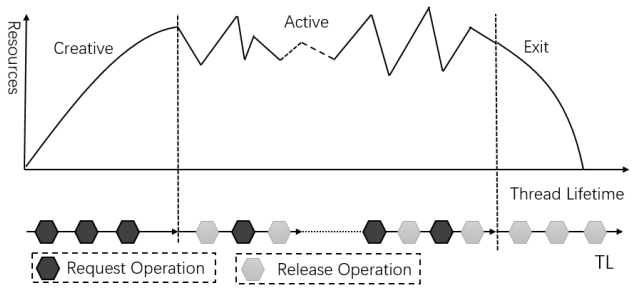


FIGURE 1. A single-thread behavior.

Second, from the perspective of the OS, the dynamic libraries of Android framework will occupy a large part of the thread's memory space. As shown in Fig. 2, the *x*-axis represents different apps, and the *y*-axis shows the ratio of shared libraries to the entire memory of a thread. Almost all the apps are written in Java and executed by the *Dalvik virtual machine (vm)* as Android provides abundant and powerful APIs in the app framework layer for designers to develop applications quickly. Therefore, the thread's memory behaviors have been encapsulated by *Dalvik vm* and the C library. Intensive interactions between apps, and the framework are reduced by the bursty traffics created by these encapsulations. However, increasingly, threads are forked in parallel to execute on multicore mobile edge computing devices.

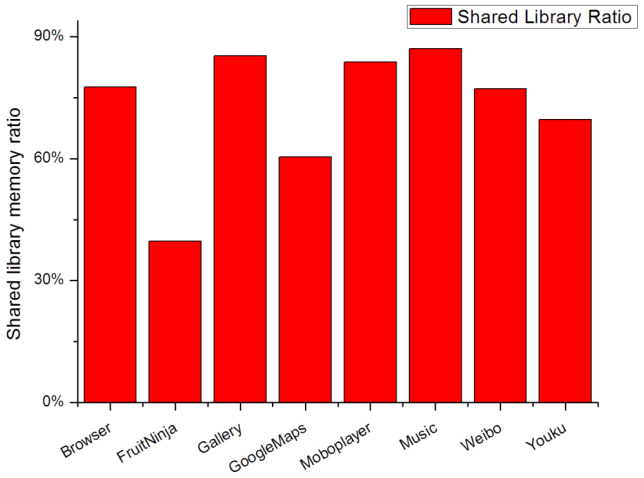


FIGURE 2. Shard library memory ratio.

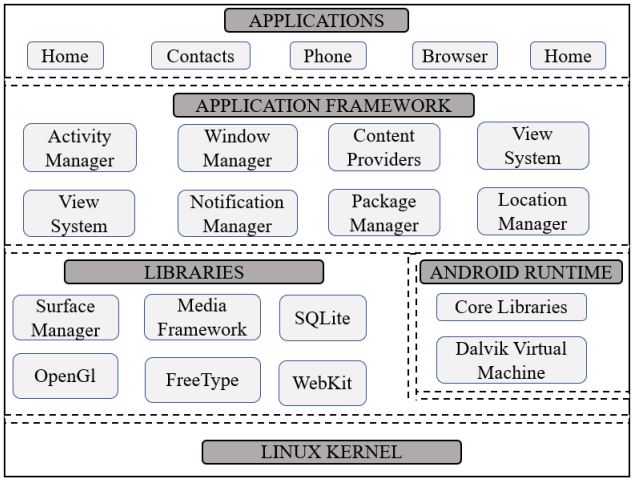


FIGURE 3. Android Architecture.

For instance, the browser app forks approximately 30 threads in half a minute, including 4 *http* worker threads, a main thread, and a DNS resolver. Therefore, these memory activities (requests and releases) are inclined to become more bursty and lead to more lock contentions. In terms of the global buddy system, each thread's frequent check on the memory management lock until it is accessible is rather time-consuming. In addition, our previous experiments revealed that a thread's waiting time for available memory increases with the number of cores.

B. THREAD-ORIENTED MEMORY MANAGEMENT LAYER

According to the thread behaviors discussed above, it can be concluded that external-fragmentation is no longer the crucial consideration when we design a new memory management framework for multicore devices. Instead, we should focus more on the inter- and intra- thread behavior described above. Thus, in the previous study, we introduced a memory management framework called MMBTB. Regrettably, there are several flaws that make it hard to adapt to fast-changing mobile devices, and it cannot meet the multiclass and high-quality demands of users. As a result, we propose a novel memory management layer called TOMML, which inherits all the advantages of MMBTB, such as taking a thread as a service object and fully exploiting the thread's behaviors on the Android platform to guide optimization. Additionally, it can fundamentally solve the shortcomings of MMBTB due to the microkernel architecture pattern that it follows. It can meet the user's requirements to select plug-ins to accomplish different optimization goals and has clear boundaries with the thread and the OS.

The Android architecture as shown above in Fig. 3 can be subdivided into five layers: Linux kernel, native libraries, the Android runtime, the framework and the complete app.

Android memory allocation is performed by the Linux kernel, native layer and framework. The main function of the Linux kernel is to score all the processes, and then update the score to the kernel, the kernel will complete the real memory recovery. Since we don't touch the kernel changes (this is one of the reasons why TOMML is transparent to threads), the native layer and framework layer are the important contents that we need to fully understand. For the native layer (libraries and Android runtime), there are two ways to develop Android apps. First, the app can be developed through the Android Java native interface (JNI). Which means that the app is developed in Java and executed by the Dalvik vm. Second, apps can be accelerated by the native development kit (NDK); for these apps, they essentially allocate and release memory with the aid of the C library's malloc and free functions. Android's JNI is implemented based on NDK functions. Therefore, all the thread's memory behaviors have been encapsulated by Dalvik vm and the C library. In other words, TOMML implements the thread's memory request and releases with the aid of native libraries. For the framework layer, Android provides powerful APIs for the developer to devise apps. As a result, most of its functions will run with Android framework support. That means that the Android framework's dynamic libraries will occupy one thread's major memory space and manifest the thread's memory behaviors. We need to consider the thread's individual behaviors and the relativeness between different threads by the framework's dynamic libraries. Meanwhile, we manage memory by calling the native layer functions instead of modifying the structure of the algorithm. Thus, TOMML is completely transparent to threads.

In light of the analysis of the preceding paragraph, the thread mainly depends on the framework and the native layer. In this, the framework is used to compose the threads (e.g., *activity manager*, *resource manager* and *content provider*), and the native layer provides a memory management interface. For example, the native layer will provide the *alloc_pages* function to meet memory requests and *free_pages* to support the release operation. Therefore, we introduced a novel memory management layer, called TOMML between the native layer and the framework layer.

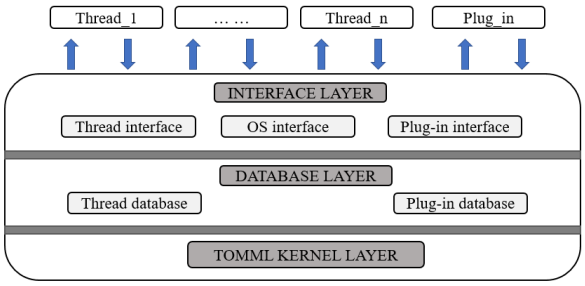


FIGURE 4. TOMML in micro-architecture.

TOMML is a novel memory management framework, which is divided into three layers and six sections as shown below Fig. 4 in the architecture diagram. The three levels include the *interface layer*, the *database layer* and the *kernel layer*, wherein the interface layer is divided into the *thread interface*, the *OS interface* and the *plug-in interface*. In addition, the database layer is divided into the thread database and the plug-in database.

- Thread interface: TOMML takes a thread as the service object which is different than the global buddy system. Therefore, we designed an interface layer for threads, which is mainly used to collect the thread's operating states, such as memory size, memory physical address range, and process identifier (PID). Particularly, we introduced two attributes: *occupy_list* is used to describe thread's in-use memory pages, and *free_list* to maintain its free memory pages. After the two attributes are automatically collected, they are saved to the thread database as shown in Fig. 5.

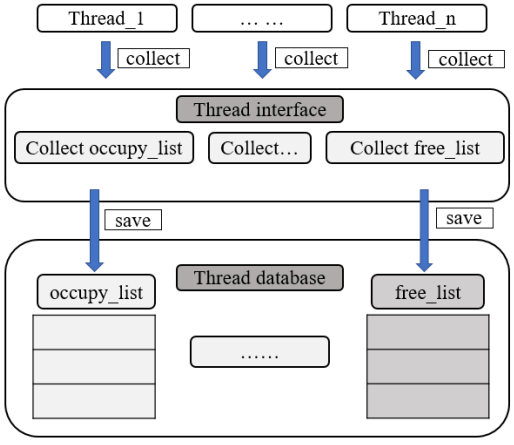


FIGURE 5. Thread interface collect the thread's operating states.

- OS interface: In the above, we explained that the memory control methods are provided through the Android native layer. Consequently, TOMML only needs to modify the implementation of these methods including the *alloc_pages* and *free_pages*. Then, there are four functions that the OS interface can use to manage memory as shown in Table 1 which is divided into intra- and inter- methods. Intra-methods include *get_page* and *release_page*, while inter-methods include *get_page_from_thread* and *get_page_to_thread*. The specific effects of the functions will be analyzed later in the example.

TABLE 1. OS interface control threads.

OS interface	
intra-methods:	get_page
	release_page
inter-methods:	get_page_from_thread
	release_page_to_thread

- Plug-in interface: This interface uses a microkernel architecture pattern, which is different from the previous MMBTB; it can meet the user's requirements for selecting plug-ins to achieve different optimization goals since it provides a unified communication protocol, and an interface to read the thread database and manipulate memory. In fact, it supplies plug-ins with access to the thread database and the OS interface.
- Thread database: This database saves the information collected by the thread interface, such as memory size, memory physical address range, PID, *occupy_list* and *free_list*. Since the information is stored as a key-value pair, its time complexity is *O(1)*.
- Plug-in database: This database stores the configuration parameters and operation log of each plug-in.
- Kernel layer: This layer provides the necessary support for other layers.

There are clear boundaries and dense interactions between each level of TOMML, as depicted in Figure 6. The gray box represents the layer name, and the white represents the main function (MF). First, the thread interface collects the thread's feature information and stores it in the thread database. After that, the plug-in manipulates the optimization methods by reading the thread information and controls the thread memory through the OS interface. In addition, users can customize plug-ins through configuring the plug-in database.

To better understand Fig. 4, we need to discuss a real example. When a page fault occurs, the Android OS will call the *alloc_pages* function to handle the memory request. Likewise, the OS also provides the *free_pages* function. Hence, TOMML only needs to modify the implementation of these functions as shown in Algorithm 1. In addition, this process is similar to the function provided by nature, since this is transparent to the applications, which means the app does not need to be modified to adapt to TOMML. In TOMML, each thread has two attributes including in-use pages and the free page. Similarly, each thread has four methods, which can be divided into inter- and intra-methods. *get_page* and *release_page* are used to get/release resources from/to a thread's free storage. Correspondingly, the *get_page_from_thread* and *get_page_to_thread* functions get/release resources from/to the other thread's free storage.

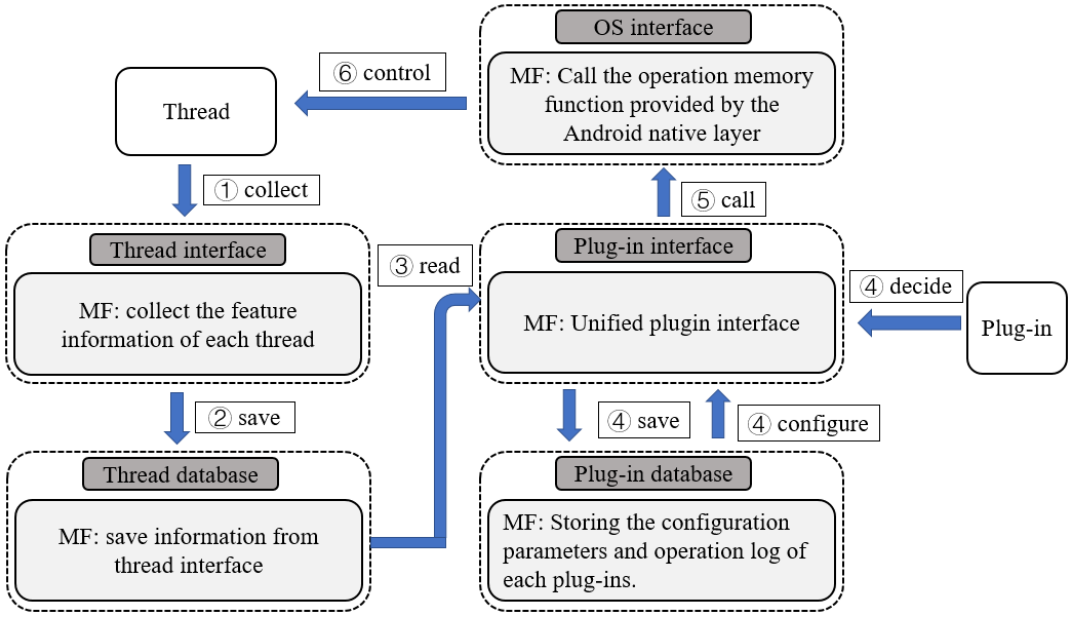


FIGURE 6. TOMML layer interaction.

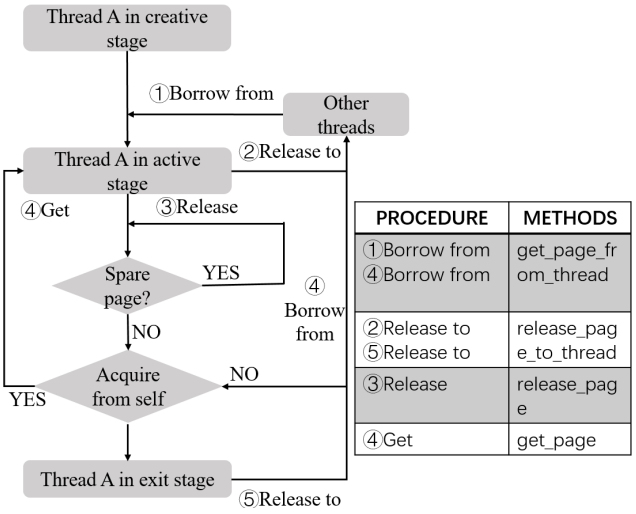


FIGURE 7. An instance of a thread.

We present an example to analyze the memory change process from the perspective of threads as illustrated Fig. 7, although this is transparent to the thread and all operations performed by TOMML. When one thread enters the create stage, it will attempt to borrow resources from other threads through inter-methods *get_page_from_thread*. Then, its active stage begins, and it will return the resources at once to ensure other thread's operation stability by *release_page_to_thread*. The only thing to note here is that the release operations are the most privileged to release resources to its own free storage by *release_page*. Corresponding to this stage, the thread will first request the pages by *get_page*; if it fails, it also has to borrow resources from other threads by *get_page_from_thread*. When it exits, resources must be released to the other threads by using the *release_page_to_thread*. These four methods are the key to the TOMML and can be used to change the two attributes (*occupy_list* and *free_list*) based on the inter- and intra- thread behavior. They can also be combined with plug-ins to achieve different optimization goals. In the next sections, we will introduce two plug-ins to optimize memory allocation and save self-refresh power consumption.

1) DEFAULT PLUG-OPTIMIZE MEMORY ALLOCATION EFFICIENCY

TOMML can speed up memory allocation-efficiency by combining with the corresponding plug-ins. In this plug-in, a new attribute called page fault frequency (PFF) is created for the thread, which is a metric to represent the thread's memory requirements [11], TOMML organizes a red-black tree based on the PFF of each thread. In addition, the leftmost node has the lowest PFF value (maximum free memory) while the rightmost node has the maximum page faults (minimum free memories). As shown in Algorithm 2, when one thread enters the create stage, there are several differences between the original and the plug-in. TOMML, which has already loaded the plug-in, will select the leftmost target thread in PFF while performing the *get_page_from_thread* method operation. Thus, the threads that hold rich free memory have higher priority to be borrowed from. By the same token, the *release_page_to_thread* method will release its rightmost with extreme memory demands. Thus, the plug-in implements the optimization goal without any changes. That is why we call it a plug-in. From this, we can discern that, since TOMML's basic structure is a red-black tree, its time complexity is $O(log\ t)$(t is the number of threads).

Algorithm 1 TOMML Functions

```
/*************************************
Input: none  Output: struct page *
*************************************/
struct page * alloc_pages(){
thread_database.pid = current_thread_
                            pid;
switch (thread_database.pid.state) {
        case CREATING_STAGE: return
            get_page_from_thread();
        case ACTIVE_STAGE: {
        if (thread_database.pid.free_
                 list == Null)
                return get_page(free_
                     list);
        else
                return get_page_from_
                     thread();
                }
        }
}
/*************************************
Input: struct page *page  Output:  none
*************************************/
void free_pages (page *page){
thread_database.pid = current_thread_
                            pid;
if(thread_database.pid.state == ACTIVE_
                                  STAGE)
        release_page(free_list);
else if(thread_database.pid.state ==
             EXITING_STAGE)
        release_page_to_thread(page);
}
```

Algorithm 2 TOMML OS Interface Functions

```
/*************************************
Input: none  Output: struct page *
*************************************/
struct page * get_page_from_thread(){
thread_database.pid = seek_min_key_
                            inrbtree;
get_page_from_thread(thread_
                             database.pid);
}
/*************************************
Input: struct page *page Output:  none
*************************************/
void release_page_to_thread (page *page)
{
thread_database.pid = seek_max_key_
                            inrbtree;
free_page_to_thread(thread_
                            database.pid);
}
```

2) EXTEND PLUG-REDUCING OPTIMIZE POWER

From previous research on mobile devices power consumption, we know that memory standby power consumption is very important to total power. There are several ways to reduce the refresh power. Liu *et al.* [12] proposed Flikker to decrease hardware reliability in an application specific manner to reduce power consumption. ESKIMO [13] is similar to Flikker and adopts a hardware mechanism to save power. However, these solutions require hardware support and are not suitable for general situations.

Currently, an increasing number of manufacturers have produced DRAM and a new feature called partial array self-refresh (PASR). PASR means that DRAM can be partially refreshed, thus further reducing the self-refresh power. Using this technique, numerous researches extend unused DRAM part's idle time by clustering applications' data together. Correspondingly, there are some classic techniques such as single/dual ended bank PASR and the bank selective PASR [14]. The most attractive methodology is the bank selective methodology since it can be signed as a self-refreshed bank individual. The plug-in we designed is based on the PASR and prolongs memory free time by clustering data together. Similar to the plug-in described in the previous paragraph, each thread is assigned one attribute called *bank_list* which is used to link other threads to the same bank. In addition, a global list *g_bank_list[0..number_banks]* is defined as *g_bank_list[i]*, which is an entry point to the bank's first thread's *bank_list*. For example, when a thread uses the borrow/release pages from/to others, it first calculates the utilization of each thread and locates the freest/fullest bank number in the *g_bank_list[0..number_banks]*. In general, there may be many threads linked to the same *g_bank_list[i]* entry. For this condition, we use the allocation efficiency to ensure the thread has the minimum/maximum page faults.

III. EXPERIMENTAL RESULTS

In this section, we will study the allocation efficiency of TOMML on the Android platform, and then prove the effectiveness of the plug-in using some intensive experiments that are introduced to determine whether the plug-in can reduce the self-refresh power of DRAM.

A. DEFAULT PLUG ON A REAL ANDROID PLATFORM

We used the vmware [15] simulator to run Android-x86 2.3 along with eight popular applications: *Youku, Weibo, Music, Moboplayer, GoogleMaps, Gallery, Fruit Ninja* and *Browser*. The experimental results are shown in Table 2.

Multicore Memory Lock Competitions: There are three independent variables in Table 2 including the application type, number of cores and memory management framework. The *x*-axis represents the different applications, and the *y*-axis

TABLE 2. Multi corecompetitions.

App / Core	Browser	Fruit Ninja	Gallery	GoogleMaps	Moboplayer	Music	Weibo	Youku
Org-Dual	0.15	0.12	0.26	0.054	0.055	0.31	0.24	0.10
Org-Quad	0.23	0.27	0.37	0.11	0.17	0.48	0.48	0.24
Org-Eight	0.31	0.30	0.38	0.21	0.18	0.62	0.56	0.32
TOMML-Dual	0.013	0.0044	0.0038	0.21	0.042	0.016	0.031	0.035
TOMML-Quad	0.013	0.0092	0.073	0.056	0.064	0.0096	0.062	0.081
TOMML-Eight	0.068	0.073	0.10	0.063	0.87	0.011	0.099	0.12

gives the lock competition ratio of the different numbers of cores under the original and TOMML memory management frameworks. It is obvious that the waiting time of the original system lock increases with the number of cores. TOMML can relief lock competitions under multicore architecture, and the lock waiting time increases from one to eight cores within 12% compared with the original system.

Distribution Efficiency: Distribution efficiency means that the speed from one thread issues memory commands (release or request) to obtain access to use them. It can be clearly seen from Table 3 that TOMML's optimized allocation efficiency for different applications ranges from 12% to nearly 60%, and as the number of cores increases, the optimization efficiency increases. There are several possible reasons for this. First, TOMML manages memory based on threads rather than the global buddy system, thus reducing lock competitions. Second, the standard buddy system takes *O(log m)* time where m is the magnitude of the memory on operating memories, whereas the time-complexity of the TOMML is *O(log n)* where *n* is the number of threads and generally much smaller than *m*. In conclusion, TOMML effectively enhances the allocation efficiency of the original system.

TABLE 3. Ratio of improving distribution-efficiency for different multi-core.

Core / App	*Single-Core*	*Dual_core*	*Quad-Core*	*Eight-Core*
Browser	0.425	0.520	0.531	0.541
Fruit Ninja	0.423	0.482	0.547	0.537
Gallery	0.352	0.542	0.573	0.562
GoogleMaps	0.462	0.558	0.563	0.560
Moboplayer	0.415	0.463	0.533	0.556
Music	0.142	0.223	0.278	0.279
Weibo	0.312	0.472	0.521	0.543
Youku	0.340	0.468	0.536	0.543

B. EXTEND-PLUGIN OPTIMIZE POWER ON A REAL ANDROID PLATFORM

Modern DDRx consists of a single package of integrated circuits. In DRAM devices, to effectively balance memory power and property, modern DRAM chips include linear and interleaved mapping modes.

- Linear Mapping Schema: In modern DRAM system's linear schema, physical address will be mapped to DRAM address in the order of *bank*, *row*, *column* and *width*, which means the bank indices will occupy DRAM address's much higher range. In this schema, an application's data would be clustered on banks as few as possible, so the self-refresh power consumption will be reduced.
- Interleaved Mapping Schema: The physical address will be mapped to DRAM address in the order of *bank*, *row*, *column* and *width*, which differs from linear mapping in that the order of *bank* and *row* is different. For access beyond the *row* size, the interleaved map accesses a different *bank*, so more *bank* can be accessed using the interleaved mapping application. This increases performance but more power consumption.

Since most modern DRAM has the PASR function, the more free areas that are in the memory, the lower the self-refresh power. That is, memory management should cluster pages into fewer banks to stay idle as long as possible. Thus, we can consider bank idle time as an important indicator to evaluate TOMML with the extend plug-in clustering effect. It is possible to quantify the refresh power consumption through bank idle time according to the micron supported DRAM power calculation method [16]. In this subsection, we discuss the clustering effects of TOMML's plugin on linear and interleaved mapping schemas.

As a control group, we chose the excellent power-aware memory allocation algorithm [17], shortened as *Ascend*, which allocates pages in order of access, filling the entire memory module before transporting to the next one. As shown in Fig. 8 (the *x*-axis is bank number and the *y*-axis shows the bank idle time ratio), the operation of the TOMML and *Ascend* memory management frameworks under different numbers of BANKs and sizes of memory use the bank idle

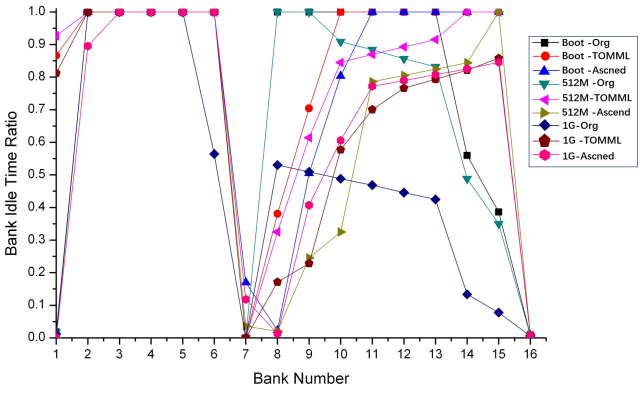


FIGURE 8. Bank idle time ratio of different allocation scenarios in linear.

TABLE 4. Improved bank idle time ratio.

	Linear Mapping Schema			Interleaved Mapping Schema		
	Ascend	TOMML	$\frac{TOMML\_Idleness}{Ascend\_Idleness}$	Ascend	TOMML	$\frac{TOMML\_Idleness}{Ascend\_Idleness}$
Boot	-0.04	0.08	0.13	66.17	70.67	0.07
512 M	-0.13	0.09	0.25	108.62	125.3	0.15
1G	0.32	0.4	0.06	108.51	125.3	0.15

time ratio as the benchmark. It can be seen that TOMML has better clustering effect than Ascend and the original framework, but the Ascend memory management plays a negative role except in the 1GB allocation scenario because in the linear mapping mode, the physical addresses will sequentially correspond to the DRAM address, and then the subsequent virtual addresses will also be mapped to subsequent banks to a large extent. Fig. 9(the x-axis is bank number and the y-axis shows the bank idle time ratio) shows the experiment in the interlaced mapping mode. After the system boots, the OS will occupy hundreds of megabytes of memory, so banks one to eight will be preassigned to the OS due to the interleaved mapping architecture, resulting in a *bank idle time* ratio close to 0. In addition, the interleaved mode will cause different physical libraries to be switched to achieve continuous virtual address access, which means that the original partner system fails to achieve good clustering effectiveness.

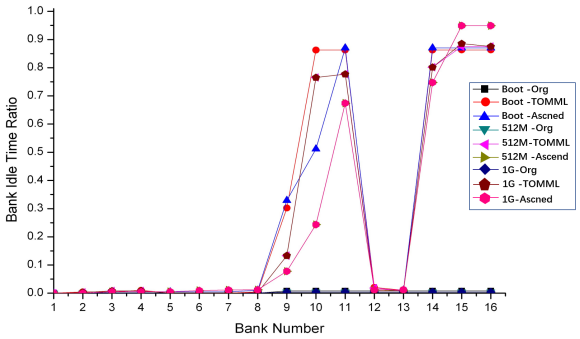


FIGURE 9. Bank idle time ratio of different allocation scenarios in interleaved.

From the above experiments, our extend plug-in can achieve better clustering efficiency in the linear mapping mode and on interleaved schema than in the control group Ascend and the original group, as shown in Table 4.

IV. RELATED WORK

Over the past ten years, the memory management of mobile devices has been studied by many scholars. Especially in recent years, since the mobile edge computing becoming increasingly popular, how to improve performance on limited power mobile devices has been a popular topic in embedded research field [18]–[24]. Android, as one of the most popular mobile devices OS, adopts a global buddy system. It is worth noting that allocation efficiency and external fragmentation are two major issues. Gerth *et al.* [5] introduced three methods to the buddy system that decreased the running time of distribution to constant worst-case time and deallocated to constant amortized time for the first two solutions and constant worst-case time for the third one. Kim *et al.* [25] presented a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them contiguously in fixed-size contiguous regions. In addition, Li *et al.* [26] proposed a global memory fragmentation quantification approach that summarizes a memory block's access pattern and measures the allocation time of different order memory blocks dynamically. Other research and analytical work [6]–[9] also exist. Such as Abdelwahab *et al.* [27] propose User-Level Online Offloading Framework (ULOOF), a lightweight and efficient framework for mobile computation offloading, which is equipped with a decision engine that minimizes remote execution overhead. Jia *et al.* [28]–[31] propose a series of new caching strategies such as Hybrid-LRU, cost aware cache replacement policy (CACRP), and dynamic adaptive replacement policy (DARP) to improve memory resources management performance. However, these previous studies ignore thread behaviors and lack target optimizations. Therefore, lots of studies [4], [32]–[35] based on app characteristics to improve the performance of mobile edge computing devices. For instance, M. Tech [36] found that out of memory (OOM) killer, activity manager service (AMS), and low memory killer (LMK) in Android kills some of the apps in low memory scenarios along with OOM killer, which will go through the memory loading cycle again and takes approximately 3-5 secs. Yin *et al.* [37]–[39] improves mobile devices performance by discovering potential relationships between users and apps. Also, other works for the hardware level focus on task behaviors, such as intra-task dependency [19], [40], to lead performance optimization.

The previous research [10] we performed differs from others since it either optimizes the algorithm's time-complexity to improve allocation efficiency, but it does not modify the structure of the algorithm to manage fragmentation. In this paper, we optimized the original MMBTB design framework based on the needs of users, which can meet the user's requirements for selecting plug-ins to achieve different optimization goals and has clear boundaries with the threads and the OS. Unfortunately, these previous similar studies work at the C shared library level of user-space; thus, they consider the thread individual behaviors and ignore the relativeness

between different threads. In addition, they are designed to optimize a particular objective and cannot meet different optimization targets. Instead, our goal is to use a microkernel architecture to optimize MMBTB for high customizability and fault tolerance.

V. CONCLUSION

In this paper, we introduced a novel memory resource management framework for edge computing devices, called TOMML. This work is motivated by the observation that the previously proposed framework MMBBT is not suitable for current users, it is hard to integrate different optimization strategies, and users cannot change optimization targets in different scenarios. Therefore, TOMML follows the microkernel architecture pattern and utilizes all the advantages of MMBTB. Experimental results from real Android systems demonstrate that our approach can improve allocation efficiency from 12% to 20%. In addition, to show our framework's plug-in interface compatibility, we make plug-in to address DRAM's self-refresh power issue in edge computing. Experiments indicate that it can increase bank idleness by 6%-25% by combining with different mapping schemas.

REFERENCES

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[3] G. Khetan, "Comparison of memory management systems of BSD, windows, and linux," *Retrieved May*, vol. 22, p. 2010, Dec. 2002.

[4] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan, "A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach," in *Proc. 36th Int. Conf. Comput.-Aided Design*. Piscataway, NJ, USA: IEEE Press, Nov. 2017, pp. 217–224.

[5] G. S. Brodal, E. D. Demaine, and J. I. Munro, "Fast allocation and deallocation with an improved buddy system," *Acta Inf.*, vol. 41, nos. 4–5, pp. 273–291, 2005.

[6] A. G. Bromley, "Memory fragmentation in buddy methods for dynamic storage allocation," *Acta Inf.*, vol. 14, no. 2, pp. 107–117, 1980.

[7] S. K. Chowdhury and P. K. Srimani, "Worst case performance of weighted buddy systems," *Acta Inf.*, vol. 24, no. 5, pp. 555–564, 1987.

[8] P. W. Purdom, Jr., and S. M. Stigler, "Statistical properties of the buddy system," *J. ACM*, vol. 17, no. 4, pp. 683–697, 1970.

[9] S. Serewa, "The improvement of the buddy system," *Theor. Appl. Informat.*, vol. 18, no. 2, pp. 133–140, 2006.

[10] Z. Zhu *et al.*, "A thread behavior-based memory management framework on multi-core smartphone," in *Proc. 19th Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, Aug. 2014, pp. 91–97.

[11] R. K. Gupta and K. A. Franklin, "Working set and page fault frequency paging algorithms: A performance comparison," *IEEE Trans. Comput.*, vol. C-27, no. 8, pp. 706–712, Aug. 1978.

[12] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 213–224, 2012.

[13] C. Isen and L. John, "Eskimo-energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2009, pp. 337–346.

[14] T. Brandt, T. Morris, and K. Darroudi, "Analysis of the PASR standard and its usability in handheld operating systems such as Linux," Intel, Santa Clara, CA, USA, Tech. Rep., 2007.

[15] M. Rosenblum, "Vmwares virtual platform," *Proc. Hot Chips*, vol. 1999, pp. 185–196, Aug. 1999.

[16] *Calculating Memory System Power for DDR3*, Micron, Boise, ID, USA, 2007.

[17] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. Newton, MA, USA: O'Reilly Media, 2005.

[18] K. Vimal and A. Trivedi, "A memory management scheme for enhancing performance of applications on android," in *Proc. IEEE Recent Adv. Intell. Comput. Syst. (RAICS)*, Dec. 2015, pp. 162–166.

[19] C. Wang *et al.*, "Architecture support for task out-of-order execution in MPSoCs," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1296–1310, May 2015.

[20] Y. Wiseman, *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. Pennsylvania, PA, USA: IGI Global, 2009.

[21] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[22] Y. Xiao, S. Nazarian, and P. Bogdan, "Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)* Mar. 2018, pp. 1387–1392.

[23] Y. Xue, Z. Qian, G. Wei, P. Bogdan, C.-Y. Tsui, and R. Marculescu, "An efficient network-on-chip (NoC) based multicore platform for hierarchical parallel genetic algorithms," in *Proc. 8th IEEE/ACM Int. Symp. Netw.-on-Chip (NoCS)*, 2014, pp. 17–24.

[24] Y. Xue, J. Li, S. Nazarian, and P. Bogdan, "Fundamental challenges toward making the iot a reachable reality: A model-centric investigation," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 22, no. 3, p. 53, 2017.

[25] S.-H. Kim, S. Kwon, J.-S. Kim, and J. Jeong, "Controlling physical memory fragmentation in mobile systems," *ACM SIGPLAN Notices*, vol. 50, no. 11, pp. 1–14, 2016.

[26] Y. Li, D. Liu, J. Zhang, and L. Long, "A quantitative approach for memory fragmentation in mobile systems," in *Proc. Int. Conf. Smart Comput. Commun.* Cham, Switzerland: Springer, 2016, pp. 339–349.

[27] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati, "Replisom: Disciplined tiny memory replication for massive IoT devices in LTE edge cloud," *IEEE Internet Things J.*, vol. 3, no. 3, pp. 327–338, Jun. 2016.

[28] G. Jia, G. Han, J. Jiang, and L. Liu, "Dynamic adaptive replacement policy in shared last-level cache of DRAM/PCM hybrid memory for big data storage" *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1951–1960, Apr. 2017.

[29] G. Jia, G. Han, J. Rodrigues, J. Lloret, and W. Li, "Coordinate memory deduplication and partition for improving performance in cloud computing," *IEEE Trans. Cloud Comput.*, to be published.

[30] G. Jia, G. Han, H. Wang, and F. Wang, "Cost aware cache replacement policy in shared last-level cache for hybrid memory based fog computing," *Enterprise Inf. Syst.*, vol. 12, no. 4, pp. 435–451, 2018.

[31] G. Jia, G. Han, H. Xie, and J. Du, "Hybrid-LRU caching for optimizing data storage and retrieval in edge computing-based wearable sensors," *IEEE Internet Things J.*, to be published.

[32] D. Gay and A. Aiken, *Memory Management With Explicit Regions*, vol. 33. New York, NY, USA: ACM, 1998.

[33] D. R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," *Softw., Pract. Exper.*, vol. 20, no. 1, pp. 5–12, 1990.

[34] H. Gao, W. Huang, X. Yang, Y. Duan, and Y. Yin, "Toward service selection for workflow reconfiguration: An interface-based computing solution," *Future Gener. Comput. Syst.*, vol. 87, pp. 298–311, Oct. 2018.

[35] H. Gao, Y. Duan, H. Miao, and Y. Yin, "An approach to data consistency checking for the dynamic replacement of service process," *IEEE Access*, vol. 5, pp. 11700–11711, 2017.

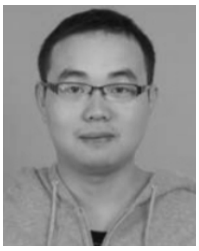
[36] R. Prodduturi, "Effective handling of low memory scenarios in android using logs," M.S. thesis, Indian Inst. Technol., New Delhi, India, 2013.

[37] Y. Yin, L. Chen, Y. Xu, and J. Wan, "Location-aware service recommendation with enhanced probabilistic matrix factorization," *IEEE Access*, vol. 6, pp. 62815–62825, 2018.

[38] Y. Yin, F. Yu, Y. Xu, L. Yu, and J. Mu, "Network location-aware service recommendation with random walk in cyber-physical systems," *Sensors*, vol. 17, no. 9, p. 2059, 2017.

[39] Y. Yin, Y. Xu, W. Xu, M. Gao, L. Yu, and Y. Pei, "Collaborative service selection via ensemble learning in mixed mobile network environments," *Entropy*, vol. 19, no. 7, p. 358, 2017.

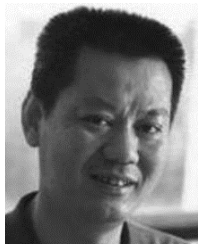
[40] C. Wang, X. Li, J. Zhang, X. Zhou, and X. Nie, "MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 2, p. 9, 2013.



ZONGWEI ZHU received the M.S. and Ph.D. degrees in computer science from the University of Science and Technology of China (USTC), in 2011 and 2014, respectively. From 2014 to 2016, he was a Research Assistant with the IOT Perception Mine Research Center, China University of Mining and Technology. From 2016 to 2018, he was a Senior Engineer with Huawei Company. He is currently a Research Assistant with the Suzhou Institute, USTC. His research interests include resource scheduling, memory, power, and operating systems.

FAN WU received the B.S. degree from the College of Agricultural Mechanization and Its Automation, Huazhong Agricultural University, in 2018. He is currently pursuing the M.S. degree with the Department of Software Engineering, University of Science and Technology of China. His current research interests include power and operating systems, artificial intelligence, and resource scheduling.

JING CAO received the B.S. degree from the College of Biomedical Engineering & Instrument Science, Zhejiang University, in 2018. She is currently pursuing the M.S. degree with the Department of Software Engineering, University of Science and Technology of China. Her current research interests include power, operating systems, artificial intelligence, and resource scheduling.



XI LI is currently a Professor of computer science and the Executive Dean of the School of Software Engineering, University of Science and Technology of China. There, he directs research programs in the Embedded System Lab, examining various aspects of embedded system with the focus on reliability, performance, availability, flexibility, and energy efficiency. He has led several national key projects in China and several national 863 projects and NSFC projects. He is a member of the ACM and a Senior Member of the CCF.

GANGYONG JIA received the Ph.D. degree from the Department of Computer Science, University of Science and Technology of China, Hefei, China, in 2013. He is currently an Assistant Professor with the Department of Computer Science, Hangzhou Dianzi University, China. He has served as a Reviewer of microprocessors and microsystems. His current research interests include the IoT, cloud computing, edge computing, power management, and operating systems.