Regular Paper

A Comparative Analysis of RTOS and Linux Scalability on an Embedded Many-core Processor

Yixiao Li[1,a) Yutaka Matsubara[1,b) Hiroaki Takada[1,c)

Received: June 7, 2017, Accepted: December 8, 2017

Abstract: Current embedded systems are usually based on real-time operating system (RTOS). In the near future, embedded systems will include parallel applications for tasks like autonomous driving, and adopt many-core processors to satisfy the performance requirements. However, traditional RTOSes are not designed for high performance applications and whether they can scale well on many-core processors remains unclear. Meanwhile, research has shown that Linux can provide good scalability for processors with tens of cores. In this paper, an experiment environment based on a traditional multi-core RTOS (TOPPERS/FMP) and an off-the-shelf 72-core many-core processor (TILE-Gx72) is presented. By a comparative analysis of RTOS based and Linux based runtime systems, several bottlenecks in RTOS are identified and the methods to avoid them are proposed. After that, the PARSEC benchmark suite is used to evaluate the performance of RTOS and Linux. The results show that the optimized RTOS runtime system tends to deliver better scalability than Linux in many cases. Therefore, we believe that traditional RTOS like TOPPERS/FMP can still be a good choice for embedded many-core processors in the near future.

Keywords: embedded system, RTOS, many-core, scalability

1. Introduction

Multi-core processors have been used in many embedded systems to satisfy the increasing performance requirements with a reasonable power consumption. Most of the embedded systems are based on a multi-core real-time operating system (RTOS) because they usually include applications with real-time constraints. Previous studies of multi-core RTOS have been mainly focused on the schedulability and response time analysis of task sets on multi-core processors [1], [2]. Meanwhile, many researchers have claimed that high-end embedded systems in the near future will also include high-performance parallel applications in order to support complex tasks like autonomous driving of vehicles [3].

Current mainstream embedded multi-core processors are not suitable for those parallel applications since they only have a small number of cores. Several off-the-shelf many-core processors aiming for future embedded systems, which contain tens (or even hundreds) of cores, have been released in recent years. Examples of those processors include the 72-core Mellanox TILE-Gx72 processor [4] and the 288-core Kalray MPPA (Multi-Purpose Processor Array) [5]. However, it remains unclear whether traditional RTOS can allow parallel applications to scale well on many-core processors, since they are not designed to provide high scalability for these applications.

In the field of high-performance computing (HPC) and cloud computing, on the other hand, the scalability problems in traditional general-purpose operating system (GPOS) have been actively researched for decades. Linux, the de facto standard kernel for HPC and cloud servers, has been considered as a bottleneck for processors with a huge number (hundreds to thousands) of cores. Some OS kernels specially designed to avoid scalability bottlenecks, such as Barrelfish [6], Corey [7], and fos [8], have been proposed. However, these kernels with new designs require different methodologies for implementing user applications (e.g., explicitly control sharing), which can make the development much more complicated than the traditional approach. Meanwhile, researchers have also shown that Linux can actually provide good scalability on many-core processors with tens of cores (or at least on a 48-core machine) and thus “there is no scalability reason to give up on traditional operating system organizations just yet” [9].

In this paper, we focus on the scalability of traditional multi-core RTOS on many-core processors with less than 100 cores. We believe those processors are very likely to become the mainstream embedded processors in the near future. The analysis is conducted by comparing an RTOS based runtime system with the well optimized Linux based one. The word “runtime system” means the OS kernel and all necessary middleware required by the user application. If RTOS shows close (or better) scalability comparing to Linux in parallel benchmark, we can say that traditional RTOSes are, at least potentially, suitable for scaling parallel applications on embedded many-core processors.

The main contributions of this paper are as follows. At first, a traditional multi-core RTOS kernel called the TOPPERS/FMP kernel [10] is ported to the 72-core TILE-Gx72 embedded many-core processor [4]. A runtime system based on the

[1] Graduate School of Information Science, Nagoya University, Nagoya, Aichi 464–0814, Japan
a) liyixiao@ertl.jp
b) yutaka@ertl.jp
c) hiro@ertl.jp

TOPPERS/FMP kernel that allows developers to execute parallel applications has been built and released [11]. We believe that it is the first publicly released open source testbed for evaluating traditional RTOS on an off-the-shelf many-core processor. The second contribution is that several bottlenecks in RTOS have been identified by comparing with Linux. Those problems actually have already been addressed in Linux and we have proposed the methods to avoid them in RTOS. Finally, we have evaluated and analyzed the scalability of RTOS and Linux by the PARSEC benchmark suite [12]. To our knowledge, no previous studies have compared the performance of traditional RTOS and Linux on a real many-core processor. The results suggest that traditional RTOS tends to deliver better performance than Linux, and thus it is still a great choice for embedded many-core systems in the near future.

The rest of the paper is organized as follows. In Section 2, we overviews the RTOS based experiment environment. In Section 3, bottlenecks in RTOS based runtime system are addressed by comparing with Linux. The performance of RTOS and Linux is evaluated and analyzed using the PARSEC benchmark suite in Section 4. Finally, this paper is concluded in Section 5.

2. Experiment Environment Overview

The biggest obstacle before going any further into the analysis is the lack of an open source testbed that allows us to evaluate a traditional RTOS on a many-core processor with high-performance parallel applications. Although an open source RTOS kernel called Erika Enterprise RTOS v3, which supports the Kalray MPPA many-core processor, has been announced by Evidence SRL, its source code has not been released yet as of this writing [13]. Therefore, we have to build an experiment environment from the ground up for further analysis.

Our experiment environment is based on the TILEncore-Gx72 [14] platform which is an off-the-shelf development board equipped with a 72-core embedded many-core processor called TILE-Gx72 and 32 GByte DDR3 memory. We have ported a multi-core RTOS kernel called the TOPPERS/FMP kernel to the TILE-Gx72 processor, and developed a runtime system based on the TOPPERS/FMP kernel with middleware required for executing parallel applications. PARSEC, a popular benchmark suite composed of multithreaded applications, is used for evaluating the performance and scalability. Experiments are performed both on the RTOS based runtime system and a Linux 4.5 based runtime system for comparison analysis. The source code of our experiment environment has been made publicly available [11].

In this section, three most basic elements in our experiment environment—the TILE-Gx72 processor, the TOPPERS/FMP kernel and the PARSEC benchmark suite—are overviewed.

2.1 TILE-Gx72 Embedded Many-core Processor

TILE-Gx72 is a 72-core processor from the TILE-Gx many-core processor family which aims for delivering high performance and energy efficiency to embedded systems and servers [4], [14]. Cores (called tiles) in TILE-Gx72 are interconnected with a 2D mesh NoC (network-on-chip) named iMesh as shown in **Fig. 1**. iMesh uses a dimension-ordered (X-Y) routing algorithm and the latency is 1 clock cycle per hop. Each core is

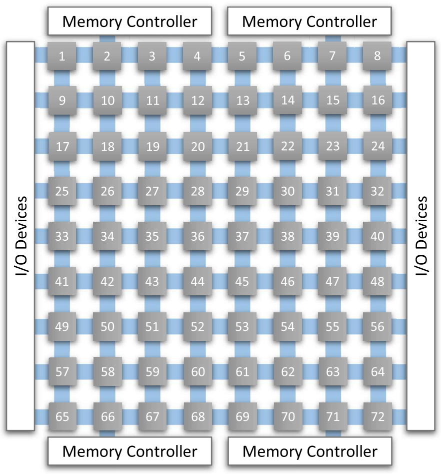

Fig. 1 iMesh NoC in TILE-Gx72 processor.

a full-featured, 64-bit processing unit working at 1 GHz, including 32 KByte private L1 instruction cache, 32 KByte private L1 data cache, 256 KByte L2 cache, and a full-blown memory management unit (MMU). There are 4 memory controllers and each of them is directly connected to a single core. Although other cores need to use iMesh network to access memory, the process is transparent to system developers.

TILE-Gx72 can provide hardware cache coherence by a technology called dynamic distributed cache (DDC). The basic idea of DDC is to use the union of all of the L2 caches as a distributed virtual L3 cache. Each physical memory address in TILE-Gx72 is associated with a home tile, which is set in the corresponding page table entry. The coherence information for a particular physical address is always tracked and maintained by its home tile. If a core wants to cache an address homed remotely into its local L2, it requests the data from the home tile instead of the memory controller. Likewise, if a core modified a L2 cache line, it will also send the new data to the home tile and the home tile will invalidate all the other copies for consistency. In this way, the L2 cache in a home tile can be viewed as a coherent L3 cache, and thus the TILE-Gx72 can be logically treated as a traditional shared memory system despite its NoC architecture.

Since a memory page is much larger than a cache line (e.g., small page is 64 KByte and L2 cache line is 256 Byte), it could by very inefficient to home an entire page with a single core in some cases. TILE-Gx72 provides a strategy called hash-for-home to maximize the average performance by effectively utilizing L2 cache and NoC bandwidth of the entire chip. If a page is marked as hash-for-home, its addresses will be distributedly homed across multiple cores (e.g., 1st L2 cache line by core 1, 2nd L2 cache line by core 2, ...). The hash-for-home strategy is recommended as the default homing policy because it can generally deliver excellent throughput.

2.2 TOPPERS/FMP Multi-core RTOS Kernel

TOPPERS/FMP kernel (or "FMP" for short) is the multi-core RTOS kernel in our experiment environment [10]. It is an open-source traditional RTOS based on the popular μITRON specification [15]. FMP is short for Flexible Multiprocessor Profile, and as its name implies, FMP has a flexible implementation which can support both symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) architectures [16].

Only the most common basic functionalities such as multitasking and time management are provided in FMP itself, since the services required by different embedded systems can vary widely. Advanced features (e.g., file system or networking) are supported by optional middleware implemented with the APIs of FMP. Therefore, when evaluating the performance of applications, we must take the whole runtime system into consideration.

FMP uses a static system design which will generate all kernel objects during compile time. Although the static design is less flexible than the dynamic design in GPOS kernel, it tends to have lower overhead because of its simplicity.

Like most multi-core RTOS kernels, the target systems of FMP are embedded systems with a small number (usually less than 10) of cores. In the original implementation of FMP, the maximum number of cores are dependent on the word length of the processor. Although it is enough for current mainstream processors, for the 64-bit TILE-Gx72 processor, only 64 of the 72 cores can be used by the kernel. We have investigated into this limitation and found it can be easily fixed. The original data type of processor affinity mask, which determines the set of cores a task can run on, is `uint_t` whose size is equal to the word length. Each bit in the mask represents a single core, and hence the number of cores is limited. We have modified the source code to use `uint32_t`$[\lceil \frac{CoreNumber}{32} \rceil]$ for these masks. Because the new data type can scale with the actual core number, FMP in our experiment environment can, at least theoretically, support many-core processors with an arbitrary number of cores.

2.3 PARSEC Benchmark Suite

The Princeton Application Repository for Shared-Memory Computers (PARSEC), is a benchmark suite for evaluating the performance of shared-memory Chip-Multiprocessors (CMPs) [12]. TILE-Gx72 can be viewed as a shared-memory CMP because all cores can coherently access the same memory system via iMesh. The most important reason for choosing PARSEC is that it focuses on next-generation applications for future CMPs [17]. Key characteristics of PARSEC are listed as follows.

Multithreaded. All applications in PARSEC have been parallelized to utilize resources of multi-core processor as possible. Various programming models such as fork-join and pipeline are covered for analyzing from different perspectives.

Emerging. PARSEC includes workloads which are considered important in the near future but not commonly used yet. It can be very helpful for figuring out to what extent a processor can meet the demands of emerging applications.

Diverse. PARSEC does not focus on some specific application domain (e.g., HPC). Instead, it includes a wide spectrum of applications, such as those for desktop and servers. Owing to this diversity, PARSEC tends to trigger more performance bottlenecks than those domain-specific benchmarks.

PARSEC 3.0 is used to evaluate our experiment environment. The original build system provided by PARSEC creates applications as executable files (e.g., ELF file in Linux) to be executed by the OS. As a static RTOS, FMP does not have a loader to execute application file at run time. We have extended the build system of PARSEC to allow an application to be created as a static library. The generated library file of PARSEC application can be linked together with the RTOS to form a single image which can be directly executed by the boot loader.

3. Runtime System Analysis and Optimization

Most user applications cannot directly run on OS kernels. Instead, runtime environments (RTEs) built above the kernels, which include necessary middleware and libraries, are required to execute them.

In this section, the characteristics of the runtime systems for FMP and Linux are analyzed. Some performance problems in the FMP based runtime system have been identified and our solution for each problem is explained and evaluated.

3.1 OS Kernels

FMP and Linux have many differences in design and implementation because they are not targeting the same systems. In this section, we will focus on comparing the factors which can have impact on the performance of applications.

3.1.1 System Design: Static vs. Dynamic

The primary function of an OS kernel is to manage hardware and software resources in the system and provide services to access them. This function can be implemented either in a static or a dynamic approach, depending on the target systems.

For most embedded systems, the required and available resources are predetermined since these systems are designed to perform some specific tasks. In FMP, kernel objects (e.g., tasks, semaphores, data queues) are statically configured, by defining them in configuration files at design time. All necessary data structures for these objects, such as control blocks, will be generated by a configurator during compile time. Each kernel object is associated with an ID for accessing with system calls.

Linux, on the other hand, has a dynamic design in order to provide the flexibility demanded by general purpose computer systems. Kernel objects are created and initialized at run time with Linux in-kernel APIs such as `kthread_create()`. Usually, these creation APIs will return a pointer of the data structure dynamically allocated for that object, which is needed for accessing.

Since the number of kernel objects in FMP is fixed at run time, they can be simply stored in arrays and referenced by using index as ID. Meanwhile, data structures of kernel objects in Linux are managed using linked lists of dynamically allocated memory blocks, because their number is considered to be unbounded. This kind of complexity introduced by a dynamic design can lead to higher overhead compared to the straightforward implementation of a static design. Previous studies have also shown that static kernel, while has limited flexibility, can achieve smaller footprint, better real-time performance and greater reliability [18], [19].

3.1.2 Basic Services and Device Drivers

From the viewpoint of application developers, an OS kernel consists mainly of basic services and device drivers, whose functionalities are usually provided as system calls.

Basic services are minimal, yet essential, elements which can be found in most OS kernels. Typical examples include memory management, time management, interrupt handling, kernel locking, and multitasking with primitives for synchronization and communication. Although both FMP and Linux support these functions, the implementations differ due to their system designs. Generally, accessing services in FMP has lower overhead while Linux is optimized to achieve better throughput and scalability.

The duties of device drivers, meanwhile, are not alike inside FMP and Linux. Since hardware of an embedded system is highly specialized for certain application domain, it is difficult for the kernel to assume what device drivers should be provided. FMP itself only includes the minimum necessary device drivers, such as those for timer, interrupt controller and serial port (if syslog is used). Other devices are supported by optional middleware components implemented on top of the kernel which are considered to be part of the RTE. Developer is required to use dedicated APIs of each middleware to control corresponding device.

On the contrary, Linux, as a GPOS kernel, must support most of the common devices, such as networking and file systems, out of the box and provide a universal approach to access them. Linux has a system call interface which has been kept stable over the decades in order to ensure portability of applications. Drivers are modules inside the kernel and interact with applications via that system call interface. More specifically, devices in Linux are abstracted as special files called device files and can be accessed with file I/O system calls like `write()`. Although this mechanism in Linux unifies interaction of device drivers using standard file operations, it also introduces overhead in the file system.

3.1.3 Task Scheduling Disciplines

Scheduler is one of the most important module in a kernel to support multitasking. Its discipline can greatly influence the scalability of an application in some cases. The objective of scheduling in multi-core systems is to determine how tasks are mapped to different cores and how CPU resources are allocated to tasks on each core, in order to meet the goals like maximizing throughput or minimizing response time. Since it is an active area of research and still has many challenges ahead [20], we will not focus on state-of-the-art methodologies in this paper. Instead, some simple but effective disciplines are used for evaluation.

In FMP, tasks in user applications are scheduled by a method called Round-robin then FIFO (RtFIFO). Tasks are assigned to cores in a round-robin fashion at creation. **Figure 2** shows an example of how 74 tasks are assigned to the TILE-Gx72 processor. The core indexes in this figure correspond to those in Fig. 1. Each core will get a task in turn until all tasks have been assigned. Tasks on the same core (e.g., task 1 and task 73 in the example) will be handled by the default fixed-priority preemptive scheduler. All tasks are set to the same priority so they will be served in FIFO fashion on each core. RtFIFO is a straightforward scheduling discipline without any dynamic load balancing mechanism. It is very effective for those parallel applications using the fork-join model with no load imbalance problems.

Linux, by default, uses the Completely Fair Scheduler (CFS) which aims to maximize overall CPU utilization. CFS periodically runs a complex load-balancing algorithm to keep the runqueues of all cores roughly balanced. It can improve the CPU

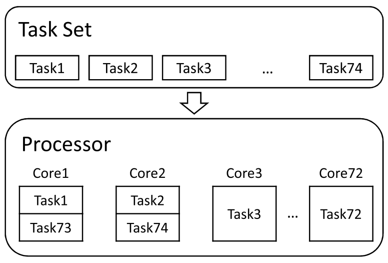

Fig. 2 Example of mapping tasks in a round-robin fashion.

utilization for applications using the pipeline model which often has load imbalance issue [21]. However, thread migrations between cores can be extremely expensive and may significantly hurt cache locality in some cases [22].

Since it is not fair to compare two runtime systems with different scheduling disciplines, we have also implemented RtFIFO method on Linux. The `pthread_attr_setaffinity_np()` function is used to bind a thread to a specific core. Threads are set to the `SCHED_FIFO` policy so they will be handled by the fixed-priority preemptive scheduler [23].

Traditionally, task assignment policy in distributed shared memory (DSM) and cache coherent NUMA (ccNUMA) systems must also consider the cost of memory access from different core. Interconnects in those systems have limited bandwidth and high latency compared to on-chip communication, which can often become a bottleneck. On the other hand, the latency of iMesh on-chip network is only 1 cycle per hop and the bandwidth of each core is over 1 terabit/s. The fast NoC allows TILE-Gx72 processor to provide the hash-for-home memory homing policy, as described in Section 2.1, which can distribute memory access across all cores in granularity of L2 cache line size. Since this policy can achieve great throughput in general and is unlikely to be a bottleneck in average case, the distance between core and memory controller is not considered by schedulers for TILE-Gx72 processor. Instead, for situations with special memory access pattern, the throughput could be further improved by controlling the home of a memory page, such as the optimization in Section 3.1.4.

3.1.4 Memory Management

Memory management is supported by most OS kernels, including static OS kernels like FMP, since there are many applications requiring the flexibility of dynamic resource management. Its function mainly consists of two parts: management of objects in kernel and management of data in user applications.

For kernel objects, the mechanisms used by FMP and Linux to manage them are quite similar. Although FMP does not support dynamic allocation of memory, it does allow user to statically define objects like memory pools for fixed-size blocks with predetermined maximum number. Therefore, dynamic management of kernel objects can be easily implemented by using the object pool design pattern [24]. In Linux, a mechanism called slab allocation [25], which is also based on object pools, is used for the efficient memory allocation of kernel objects.

For processor like TILE-Gx72 which uses paged MMU, dynamic allocation of data for user applications technically means associating page table entries with free physical memory. In FMP, all page tables are statically generated with unused physical memory defined as a pool which can be accessed from user application. That is to say, from the viewpoint of pages, all free physical memory has already been allocated to user application in advance. The preallocated pool of free memory can be logically managed by a memory allocator middleware and there is no need to dynamically change page tables at run time. Meanwhile, Linux has an extremely complex implementation for page table management with advanced features including demand paging and swapping [26]. Modifying page tables dynamically could be a bottleneck in some cases and has been actively studied and optimized over many years [27]. Consequently, if an application frequently requests the memory management services from kernel, it may have better performance on FMP than Linux.

As mentioned in Section 2.1, hash-for-home strategy for pages can provide higher throughput in general. However, there do exist some situations where homing an entire page with a single core is likely to deliver better performance due to data locality. For example, the stack of a task is usually considered as a private memory area which will be heavily used by its owner but barely be accessed from other tasks. A task will have to frequently communicate with other cores via iMesh if its stack is distributedly homed by hash-for-home strategy. Memory manager in Linux includes an optimization which will home data structures like stacks locally, in order to improve performance for these cases.

We have also introduced a similar optimization to the process of page table generation in FMP. Firstly, each core has its own section (e.g., "`.local_cached_prc1`" for core 1) to store local data. While generating source code for kernel objects, data structures such as task stacks, interrupt stacks, task control blocks and processor control blocks will be placed into the section of its local core, using the section attribute supported by the compiler. At last, page table entries for each section will be generated with home set to the corresponding core. The effectiveness of this optimization is difficult to quantify since it is heavily dependent on how these data structures are specifically used by application. An example of its effect on spinlocks will be shown in Section 3.1.5.

3.1.5 Kernel Locking

Locking is an essential mechanism in multi-core OS kernels to support inter-core synchronization and communication. The granularity of locking model in a kernel has a great impact on the overall throughput, since almost all the shared resources are required to be protected by locks.

In FMP, there are three levels of granularity to choose from: giant lock (`G_KLOCK`), processor lock (`P_KLOCK`) and fine-grained lock (`F_KLOCK`). In giant lock mode, all kernel objects share a solitary global lock and thus kernel services like system calls can only be accessed serially. This mode requires least memory space but has highest resource contention. In processor lock mode, kernel objects on the same core share a single lock and thus requests of kernel services on different cores can be processed concurrently. In fine-grained lock mode, each kernel object has its own lock so kernel services can be provided as parallel as possible. This mode has the lowest resource contention but will use much more memory than other modes if there are a large number of kernel objects. Since our experiment environment has lots of available memory, we uses the fine-grained lock mode to maximize parallelism.

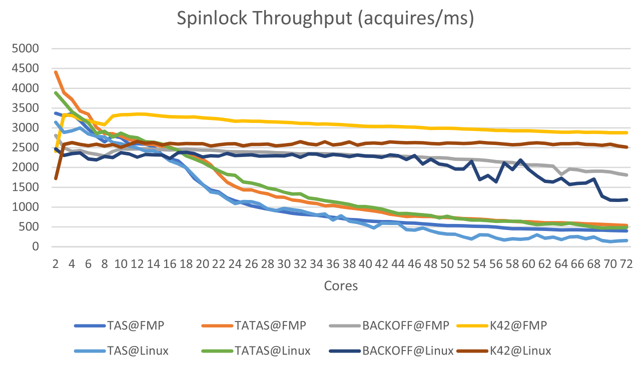

Fig. 3 Spinlock throughput of different implementations.

In previous versions of Linux, there used to be a Big Kernel Lock (BKL) which is just like the giant lock in FMP. From Linux 2.6.39, the BKL has been completely removed and replaced by a fine-grained locking scheme [28]. The granularity of locking in current Linux kernel is a bit coarser than the fine-grained lock mode in FMP because Linux is more complex and includes many components consisting of multiple kernel objects.

Spinlock is the most basic primitive for locking and advanced locks like semaphores are implemented using spinlock APIs. Therefore, spinlock implementation in kernel can heavily influence the scalability of the whole system. Typically, test-and-set (TAS) spinlock and one of its optimized variant called test-and-test-and-test (TATAS) spinlock [29] are used in RTOS for embedded systems like FMP. TAS and TATAS spinlocks have extremely simple implementations (usually several lines of C code) and only require the hardware to support a single atomic operation—the test-and-set instruction—which can be found in most multi-core processors. Although they can provide excellent performance on embedded systems with only a few of cores due to the simplicity, the throughput can dramatically collapse on many-core processors because the implementations are not scalable [30]. In Linux, a ticket spinlock with exponential backoff (hereinafter "BACKOFF spinlock") is used for TILE-Gx72 processor. The BACKOFF spinlock is not scalable either but considered to have better throughput than TAS and TATAS spinlocks [29].

In order to evaluate the scalability of different spinlocks on TILE-Gx72, we have implemented BACKOFF spinlock in FMP and TAS and TATAS spinlocks in Linux. We have also implemented K42 spinlock [31], which is a variant of the scalable MCS spinlock with compatible APIs, to compare the difference between scalable and non-scalable implementations.

We have measured the throughput of different spinlocks on FMP and Linux with a microbenchmark, and the result is shown in **Fig. 3**. In our microbenchmark, each core will loop for 100,000 times to acquire a shared spinlock, read and write 4 shared cache lines, and then release the lock. The throughputs of TAS and TATAS spinlocks are indeed better than others when the core number is small (less than 6) but will decrease rapidly. BACKOFF spinlocks have a much slower speed of throughput decreasing than TAS and TATAS and can still provide a relatively acceptable performance when all cores are used. K42 spinlocks can maintain good scalability as core number increases, but have the worst performance when less than 3 cores compete for the shared

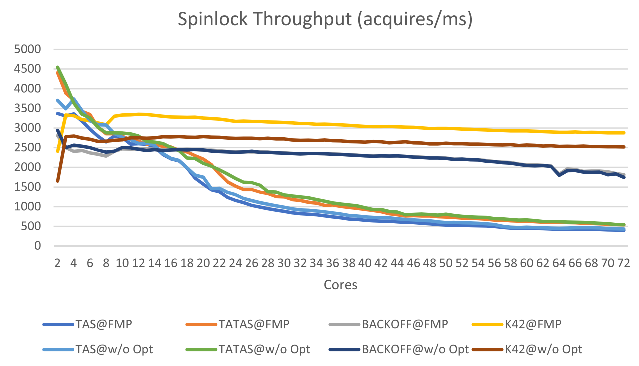

Fig. 4 Spinlock throughput in FMP with and without page optimization.

resource. Generally, FMP shows better performance than Linux when the same spinlock implementation is used. We believe that it is mainly because background system tasks in Linux can sometimes preempt the execution of our microbenchmark application.

When comparing FMP with Linux, the page table optimization introduced in Section 3.1.4 is always enabled for the sake of fairness. We have also measured the throughput of spinlocks for FMP when the optimization is disabled and the result is shown in **Fig. 4**. It can be seen that this optimization barely has any effect on TAS, TATAS and BACKOFF spinlocks. However, it does increase the scalability of K42 spinlocks for about 20%. It is because that K42 is the only implementation in them which actively uses the task stacks (for spinlock queue node).

3.1.6 Getter Function for Thread ID

One of the easiest and most effective methods to enhance scalability is replacing a globally shared data structure with a thread-local one. Since each thread has its own copy for that variable, they do not need to use a lock for synchronization and cache line contention can also be reduced.

In order to access a thread-local variable, a thread must be aware of its identifier. Thread libraries and OSes usually provide a function to get the ID of a thread (e.g., `pthread_self()` in pthread, `gettid()` in Linux and `get_tid()` in FMP). If a program frequently accesses thread-local data, the overhead of the getter function will become important. In Linux (and its pthread library) for TILE-Gx72, a special register named `tp` is used to store the thread ID, and getter functions can just read and return the value in `tp` register with a few instructions. Meanwhile, the getter function in FMP has a generic and safe implementation. It does not depend on any detailed hardware specification, but instead, has to lock the core before reading the value from memory. Further, an error code will be returned if it is not called from the task context. Consequently, the default getter implementation in FMP has larger overhead than the optimized one in Linux.

To avoid the bottleneck caused by getter, we have optimized the getter function in FMP with two methods: the `FAST` method and the `SPR` method. The `FAST` implementation is still hardware independent but has all error checking code in the getter function removed. Developer is responsible for calling this `FAST` function properly. The `SPR` implementation is, on the other hand, using a special register just like the optimization approach in Linux. It only works on TILE-Gx72 but has the minimal overhead. An example of how different implementations can affect the throughput of a scalable middleware will be shown in Section 3.2.5. In further performance comparison with Linux, FMP always uses the `SPR` implementation of the getter function for thread ID.

3.2 Middleware in Runtime Environment

In this section, we analyze necessary middleware in FMP and Linux used to execute the PARSEC applications.

3.2.1 In-memory File System

Most applications in PARSEC use files for data input and output. The speed of storage devices can always be a bottleneck but it is not related to the runtime system. In order to rule out the factor of disk I/O, in-memory file systems are usually used when evaluating the performance of applications. In fact, the development board in our experiment environment does not even have any non-volatile storage such as hard disk drive.

For Linux, the in-memory `tmpfs` file system is used by default. For FMP, we use an in-memory file system middleware called `RAMfs`. The `fmpfs` and `RAMfs` have almost the same throughput since both of them are just redirecting the file I/O system calls to simple memory operations.

3.2.2 POSIX Thread Library

PARSEC applications use POSIX threads (pthreads) for multi-threading functionality. As defined in its specification, a library for pthreads mainly consists of functions for thread management (e.g., creating, joining) and synchronization (e.g., mutex, condition variable, lock, barrier).

For Linux which is a POSIX-conformant kernel, a library called Native POSIX Thread Library (`nptl`) is used. For FMP which is not based on the POSIX standard, we have implemented a compatible layer called `POSIX4FMP` to provide pthreads support. Both `nptl` and `POSIX4FMP` are basically the wrapper for system calls. Although they are provided in libraries as part of the RTE, they are more like extension of the OS kernel. Therefore, their performance are actually dependent on the kernels.

3.2.3 C Standard Library

The C standard library (or “libc”) is the standard library for C language which provides the most commonly-used functions as specified in the ANSI C standard [32]. All applications in PARSEC heavily depend on it so its implementation can hugely influence the performance.

The SDK of TILE-Gx72 provides two libc implementations: Newlib [33] for RTOS and bare metal environment, and the GNU C library (glibc) [34] for Linux. Newlib and glibc are implemented with different system call interfaces, which means, unfortunately, we cannot use the same libc for FMP and Linux.

A libc mainly consists of functions for string handling, I/O operations, mathematical computations, and memory allocation. String handling functions are just trivial so Newlib and glibc have very close performance for them. I/O operation functions for files are basically wrappers of relative system calls in both Newlib and glibc. Other I/O functions like `printf()` are rarely used by PARSEC applications. Therefore, functions for mathematical computations and memory allocation are the most critical functions in libc which can greatly affect the performance.

In fact, both Newlib and glibc provide the mathematical functions as a separate library called `libm`. These libraries barely

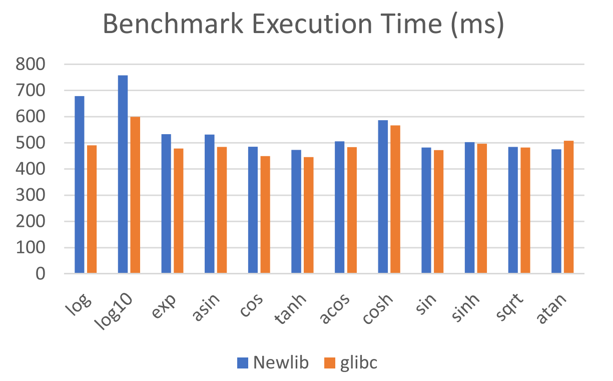

Fig. 5 Benchmark execution time of different mathematical libraries.

depend on system calls and thus we can effortlessly use them in both FMP and Linux. The mechanism to replace the functions of memory allocation is also provided by Newlib and glibc. If a library of memory allocator is linked before libc, the default memory allocator inside libc will just be overridden.

In performance evaluation, we use the same `libm` and memory allocator for FMP and Linux, which will be described later. Hence, the influence by different libc libraries is very small.

3.2.4 Mathematical Library

The mathematical library (`libm`), as mentioned above, can have an impact on the performance, especially for those compute-intensive applications.

Usually, RTOS like FMP just uses the `libm` in Newlib because Newlib is basically the only libc targeting for embedded systems. However, we have found that although glibc only works on Linux, the `libm` in glibc can actually be easily used in FMP.

We have measured the performance of mathematical functions in Newlib and glibc with a microbenchmark, and the result of some functions with relatively large difference is shown in **Fig. 5**. For logarithm functions `log()` and `log10()`, glibc is about 30% faster than Newlib. The `atan()` in glibc is about 6% slower. For most other functions, glibc shows slightly (≈5% on average) better performance than Newlib. If an application frequently calls functions like `log()`, the `libm` in Newlib which is currently used by most embedded systems, can become a bottleneck.

3.2.5 Scalable Memory Allocator

The performance of memory allocator has been studied for decades [35], [36] because it can be very important for applications requiring dynamic memory management.

In current RTOS based embedded systems, usually two memory allocators are used: the default memory allocator in Newlib and the TLSF memory allocator [36]. The allocator in Newlib is an implementation of Doug Lea's Malloc (dlmalloc) [37] which aims for maximizing average performance and minimizing fragmentation. TLSF, meanwhile, has higher fragmentation but can guarantee bounded response time which is essential for hard real-time applications. However, both TLSF and Newlib use global locking for thread safety and thus they are not scalable.

In Linux, glibc has already used a scalable memory allocator called ptmalloc by default. Since ptmalloc is implemented with Linux API in mind, we cannot use it in FMP.

To our knowledge, there is no scalable memory allocator targeting for embedded systems currently. In order to avoid the bottleneck caused by non-scalable allocator, we have ported a scalable allocator called Hoard [38] to FMP. Hoard is designed to be cross-platform but only works on Linux, Solaris, Mac OS X, and Windows out of box. It is developed in C++ while most RTOS kernels are in C language. We have created a C language interface for RTOSes to use Hoard. Our interface only requires 8 functions, which can be easily implemented in most RTOSes, such as acquiring or releasing a lock, to be provided.

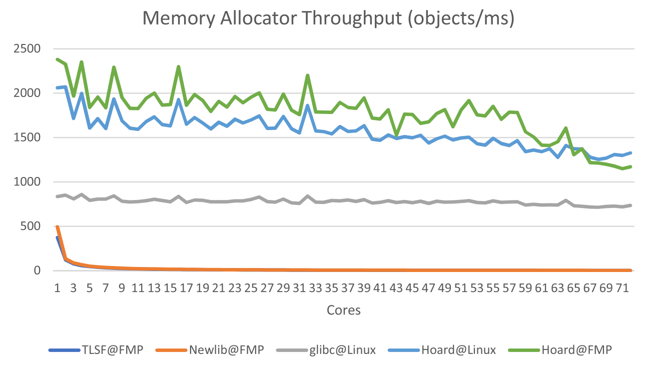

Fig. 6 Memory allocator throughput of different implementations.

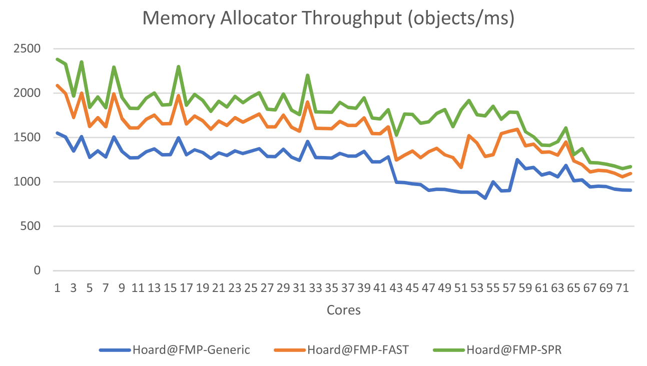

Fig. 7 Memory allocator throughput in FMP with different optimizations.

Figure 6 shows the throughput of different memory allocators. Both TLSF and Newlib will collapse quickly since they are not scalable. Allocator in glibc scales excellently but the absolute throughput is only about half as good as Hoard. Hoard shows similar scalability in FMP and Linux and has the best throughput. In most cases, Hoard in FMP has better absolute performance than Linux, and we believe it is because, as mentioned in Section 3.1.4, FMP need not to update pages at run time like Linux.

Hoard can also be a good example to evaluate the effectiveness of different optimization methods for the thread ID getter function described in Section 3.1.6, since it has a scalable implementation which actively uses thread-local variables. **Figure 7** shows the throughput of Hoard with different getter function implementations. All of them have similar trends in scalability but the generic one without any optimization shows the worst performance. The hardware-independent `FAST` optimization has about 40% better throughput than the generic. The `SPR` optimization using a special register on TILE-Gx72 has about 15% better throughput than the `FAST`. It is suggested that our two optimization methods can be very effective for scalable implementations like Hoard.

4. Performance Evaluation

In this section, we will evaluate and analyze the scalability of

Table 1 Runtime system settings for FMP-Base and Linux-Base.

	FMP-Base	Linux-Base
Spinlock	BACKOFF	
Scheduling	RtFIFO	
File System	RAMfs	tmpfs
POSIX Thread	POSIX4FMP	nptl
C Standard Library	Newlib	glibc
Mathematical Library	libm from glibc	
Memory Allocator	Hoard	

FMP and Linux using four representative PARSEC applications with different runtime system settings.

4.1 Runtime System Settings

Four runtime system settings, FMP-Base, Linux-Base, FMP-K42 and Linux-CFS, are used to run applications for evaluation.

FMP-Base and Linux-Base settings, which are shown in **Table 1**, are used to analyze the performance difference caused by different kernel implementations. For other elements in the runtime system, the same conditions are used as possible. Although file systems are different, both of them are in-memory file system and, hence, have almost the same throughput. As discussed in Section 3.2.3, the most important functions in C standard library which can have significant impact on performance are those functions for mathematical operations and memory allocation. Since the same mathematical library and memory allocator are used, the influence of different C standard library implementations is very small for PARSEC. POSIX thread libraries are considered as extension of the OS kernel so we should always analyze them together with the kernel. Therefore, FMP-Base and Linux-Base settings are suitable for comparing different OS kernels.

FMP-K42 is the setting that replaces BACKOFF spinlock in FMP-Base with K42 spinlock. By comparing FMP-Base and FMP-K42, we can evaluate how different spinlock implementations can affect the performance. Linux-CFS is the setting that replaces RtFIFO scheduling in Linux-Base with CFS. By comparing Linux-Base and Linux-CFS, we can evaluate how different scheduling disciplines can affect the performance.

4.2 Blackscholes

Blackscholes is a mathematical finance application that calculates the price of options with the Black-Scholes partial differential equation. It is the simplest application in PARSEC and has negligible communication between threads. It is a data parallel application using the fork-join model.

Figure 8 shows the scalability of blackscholes. Both FMP-Base and Linux-Base can scale well but FMP-Base is a bit better. K42 spinlock and CFS have negligible influence on this application.

In order to analyze the reason of performance difference in FMP and Linux, we have broken down the execution time of the critical path for blackscholes. The critical path is the longest execution path of a parallel application which can determine the performance of the whole application. As a fork-join application, the critical path of blackscholes can be easily found. The execution time is broken into four parts—work for computing, sync for synchronization and communication, file for file I/O operations and allocator for memory allocation—as shown in **Fig. 9** (with some

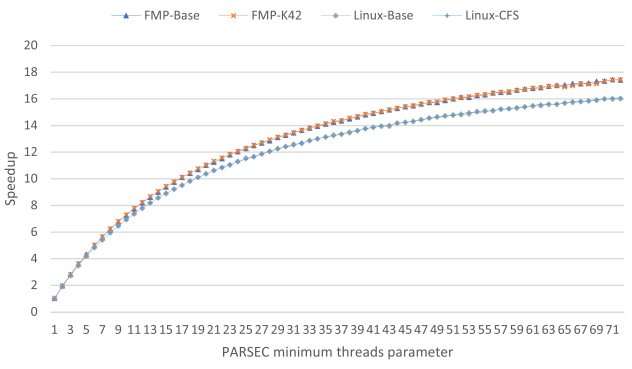

Fig. 8 Scalability of blackscholes with different runtime system settings.

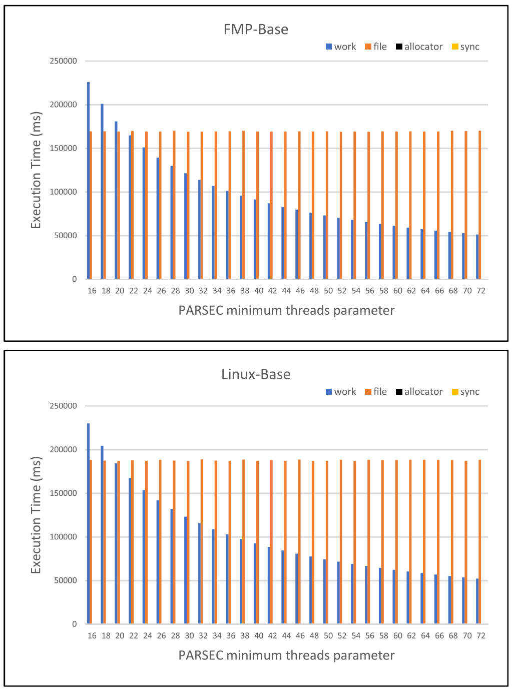

Fig. 9 Breakdown of blackscholes execution time.

unimportant data omitted to make the figures easy to read). We can see that work in blackscholes scales well but execution time for file does not scale at all. In fact, blackscholes handles file serially and this bottleneck has already been reported by previous study [39]. FMP can deliver better performance for blackscholes because the file operations in FMP cost less time than in Linux.

Although the in-memory file system modules in FMP and Linux have almost the same throughput, the overhead costs for accessing them are different. For example, in Linux, as described in Section 3.1.2, device drivers are abstracted as special files and also use the standard file operation system calls to interact with user applications. Meanwhile, those functions in FMP are only for accessing the in-memory file system. Therefore, file operations in FMP can be faster than in Linux due to the simplicity.

4.3 Swaptions

Swaptions is an application which uses Monte Carlo simulation to compute the prices of swaptions. It is compute-intensive and has a little synchronization (e.g., tens of locks) between threads. It is a data parallel application using the fork-join model.

Figure 10 shows the scalability of swaptions. All the run-

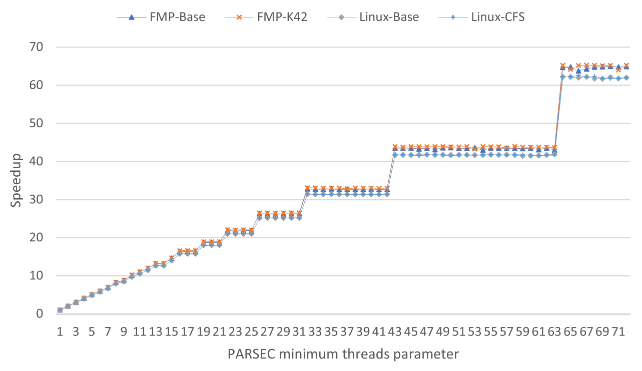

Fig. 10 Scalability of swaptions with different runtime system settings.

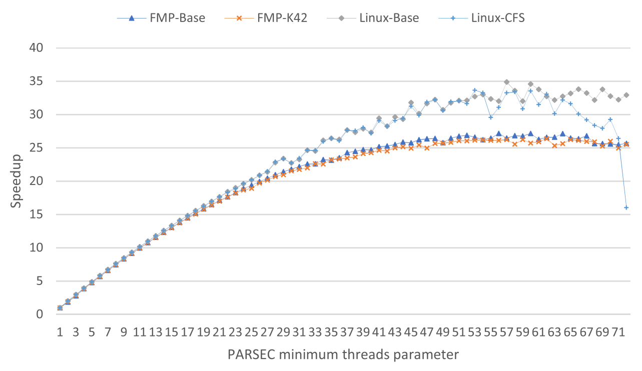

Fig. 11 Scalability of streamcluster with different runtime system settings.

time system settings have almost the same scalability. FMP does have a very slightly (about 5%) better performance than Linux in some cases, but the differences are too small for analysis. It is suggested that FMP and Linux have very close scalability for compute-intensive applications with little communication.

4.4 Streamcluster

Streamcluster is an application to solve the online clustering problem which is widely used in data mining. It has more than 100,000 barriers for synchronization between threads. It is a data parallel application using the fork-join model.

Figure 11 shows the scalability of streamcluster. Linux-Base has much better scalability than FMP-Base. The scalability of Linux-CFS is very close to Linux-Base when thread parameter is less than 64, but it will decline quickly at last. The difference between FMP-Base and FMP-K42 is ignorable.

Figure 12 shows the breakdown of execution time. The work in all settings scales well but the time for sync differs a lot. Previous study has shown that the inefficient barrier implementation in streamcluster can be a bottleneck [40]. Streamcluster uses a spin-then-block strategy for barrier by default. However, streamcluster does not have load imbalance problem, which means spinning can be much better than blocking in most situations. The barrier in streamcluster is implemented using mutexes and conditional variables provided by the POSIX thread library. In fact, the mutex implementation in Linux also uses a spin-then-block strategy. That is to say, although FMP and Linux use the same source code for barriers in streamcluster, the barriers in Linux can actually spin longer than FMP. Consequently, Linux can show better scalability than FMP. The collapse of scalability in Linux-CFS is caused by the load balancing. Since streamcluster is not imbalanced, load balancing cannot increase its performance at all. On the contrary, expensive thread migrations can harm the performance of synchronization.

To rule out the difference caused by the inefficient barriers, we have measured the scalability again with spin barriers. As shown in **Fig. 13**, the scalability is greatly improved with spin barriers and the trends for all the settings have become very similar.

4.5 Dedup

Dedup is an application for multithreaded data compression. It is communication intensive and uses the pipeline model. More

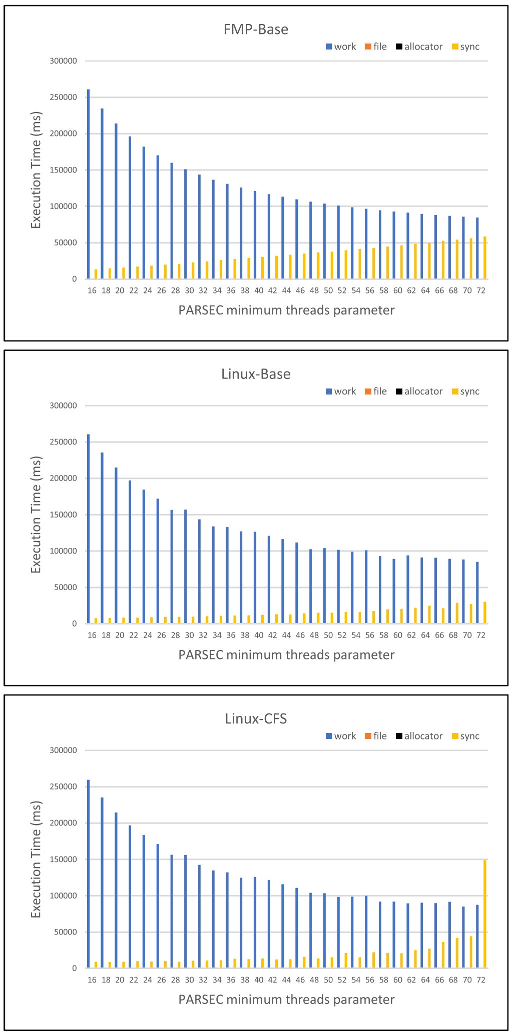

Fig. 12 Breakdown of streamcluster execution time.

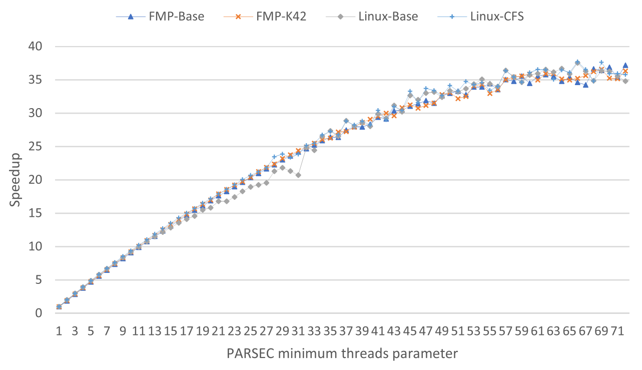

Fig. 13 Scalability of streamcluster with spin barriers.

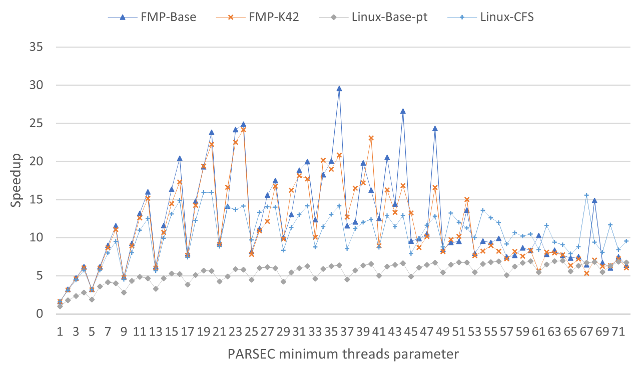

Fig. 14 Scalability of dedup with different runtime system settings.

than 150,000 locks and thousands of conditional variables are used for synchronization. Unlike those fork-join applications, the number of threads running simultaneously in dedup can be as 3 times as the thread parameter.

We have found that Linux-Base setting cannot be used for dedup due to a CPU starvation caused by spinlock. We believe that it is because Hoard for Linux is not designed to work under fixed-priority preemptive scheduling. For fork-join applications, it is fine since there is only one worker thread for each core. For pipeline application like dedup, if a thread spins to wait a lock held by another thread on the same core, the starvation can happen. Therefore, we use another setting called Linux-Base-pt for dedup. In Linux-Base-pt, the Hoard memory allocator in Linux-Base is replaced by the ptmalloc from glibc.

Figure 14 shows the scalability of dedup. Although dedup uses many spinlocks for synchronization, FMP-K42 has worse performance than FMP-Base. It is because that most spinlocks in dedup are used to protect elements in a hash table. Typically, an element in that hash table will not be accessed by multiple threads at the same time. As described in Section 3.1.5, K42 actually has the worst throughput if there are less than 3 threads acquiring the same lock. The Linux-Base-pt shows the worst scalability and it is suggested that the ptmalloc memory allocator has become a bottleneck. The scalability of FMP-Base is about twice as good as Linux-CFS when the thread parameter is smaller than 50. Linux-CFS has the best performance at last.

The analysis of scalability for pipeline application is very complex because it is not determined by a single path like fork-join applications [21]. Though, we can still shed some light by breaking down the execution time. We have summed up the average execution time of every pipeline stages and the result is shown in **Fig. 15**. The time in file and allocator is much short in FMP-Base, which should be the main reason why FMP has better scalability in most cases. The load balancing mechanism in Linux-CFS keeps the sync time to a relatively small value, and we believe that is why Linux-CFS can provide better performance at last. However, we can also see higher work time in Linux-CFS and it is suggested that thread migrations can slow down the execution.

5. Conclusion

We have presented an experiment environment based on the TOPPERS/FMP kernel and the 72-core TILE-Gx72 processor. It is the first, to our knowledge, publicly released open source

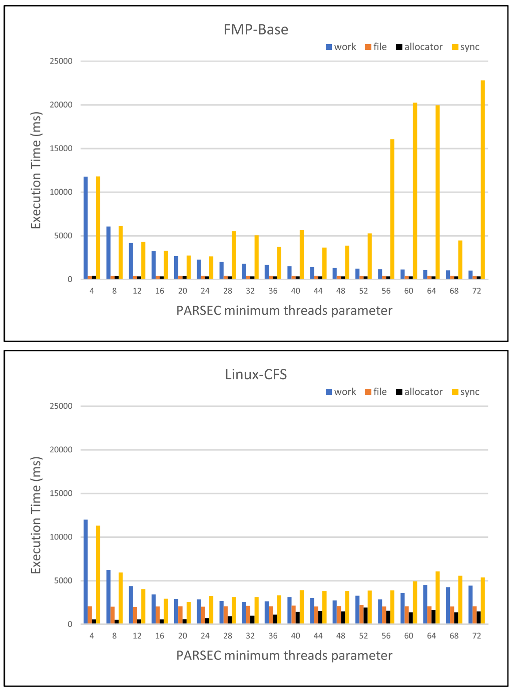

Fig. 15 Breakdown of dedup execution time.

testbed for evaluating traditional multi-core RTOS on an off-the-shelf embedded many-core processor. By a comparative analysis of RTOS based and Linux based runtime systems, we have identified several bottlenecks in traditional RTOS, such as non-scalable spinlock and memory allocator implementations. These bottlenecks have already been addressed in Linux and the methods to avoid them in RTOS are proposed in this paper. Finally, performance evaluation on RTOS and Linux is performed using the PARSEC benchmark suite. The results show that, although traditional RTOS is not designed for executing high performance applications, it can deliver better scalability than Linux in many cases, with the optimization methods in this paper. Therefore, since previous study has shown that Linux can scale well on many-core processors with tens of cores, we believe that traditional RTOS like TOPPERS/FMP can also be a good choice for embedded many-core processors.

In the future, we are planning to further evaluate the scalability of RTOS with applications using different parallel programming models such as OpenMP and message passing. We also want to investigate the load balancing on RTOS since it can be important for pipeline applications.

References

[1] Saidi, S., Ernst, R., Uhrig, S., Theiling, H. and de Dinechin, B.D.: The shift to multicores in real-time and safety-critical systems, *2015 International Conference on Hardware/Software Codesign and System Synthesis* (*CODES + ISSS*), pp.220–229, IEEE (2015).

[2] Ungerer, T., Bradatsch, C., Gerdes, M., Kluge, F., Jahr, R., Mische, J., Fernandes, J., Zaykov, P.G., Petrov, Z., Böddeker, B., et al.: parMERASA–multi-core execution of parallelised hard real-time applications supporting analysability, *2013 Euromicro Conference on Digital System Design* (*DSD*), pp.363–370, IEEE (2013).

[3] Pinho, L.M., Quinones, E., Bertogna, M., Marongiu, A., Carlos, J.P., Scordino, C. and Ramponi, M.: P-socrates: A parallel software framework for time-critical many-core systems, *2014 17th Euromicro Conference on Digital System Design* (*DSD*), pp.214–221, IEEE (2014).

[4] Mellanox Technologies Website: Overview of the 72-core TILE-Gx72 processor, Mellanox Technologies (online), available from

⟨http://www.mellanox.com/page/products_dyn?product_family=238&mtag=tile_gx72⟩ (accessed 2017-05-29).

[5] KALRAY Corporation Website: MPPA: The Supercomputing on a chip solution, KALRAY Corporation (online), available from ⟨http://www.kalrayinc.com/kalray/products/⟩ (accessed 2017-05-29).

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp.29–44, ACM (2009).

[7] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M.F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y.-H., et al.: Corey: An Operating System for Many Cores, *OSDI*, Vol.8, pp.43–57 (2008).

[8] Wentzlaff, D. and Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores, *ACM SIGOPS Operating Systems Review*, Vol.43, No.2, pp.76–85 (2009).

[9] Boyd-Wickizer, S., Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., Zeldovich, N., et al.: An Analysis of Linux Scalability to Many Cores, *OSDI*, Vol.10, No.13, pp.86–93 (2010).

[10] TOPPERS Project Website: Introduction to the TOPPERS/FMP kernel, TOPPERS Project Inc. (online), available from ⟨https://www.toppers.jp/en/fmp-kernel.html⟩ (accessed 2017-05-29).

[11] Li, Y.: TOPPERS/FMP on TILE-Gx Many Core Processor, ERTL (Embedded Real-Time Laboratory) (online), available from ⟨https://github.com/fmp-mc⟩ (accessed 2017-05-30).

[12] PARSEC team: The PARSEC Benchmark Suite, Princeton University (online), available from ⟨http://parsec.cs.princeton.edu⟩ (accessed 2017-05-30).

[13] Erika Website: Erika Enterprise RTOS v3, Evidence SRL (online), available from ⟨http://www.erika-enterprise.com⟩ (accessed 2017-05-31).

[14] Mellanox Technologies Website: TILEncore-Gx72 Intelligent Application Adapter, Mellanox Technologies (online), available from ⟨http://www.mellanox.com/page/products_dyn?product_family=231⟩ (accessed 2017-06-01).

[15] TRON Forum Website: μITRON Specification, TRON Forum (online), available from ⟨http://www.tron.org/specifications⟩ (accessed 2017-05-30).

[16] Tomiyama, H., Honda, S. and Takada, H.: Real-time operating systems for multicore embedded systems, *International SoC Design Conference, ISOCC '08*, Vol.1, pp.I–62, IEEE (2008).

[17] Bienia, C., Kumar, S., Singh, J.P. and Li, K.: The PARSEC benchmark suite: Characterization and architectural implications, *Proc. 17th International Conference on Parallel Architectures and Compilation Techniques*, pp.72–81, ACM (2008).

[18] Li, Y., Ishikawa, T., Matsubara, Y. and Takada, H.: A platform for LEGO mindstorms EV3 based on an RTOS with MMU support, *OSPERT 2014*, pp.51–59 (2014).

[19] Hoffmann, M., Dietrich, C. and Lohmann, D.: Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience, *GI-Jahrestagung*, pp.2562–2576 (2013).

[20] Singh, A.K., Shafique, M., Kumar, A. and Henkel, J.: Mapping on multi/many-core systems: survey of current and emerging trends, *Proc. 50th Annual Design Automation Conference*, p.1, ACM (2013).

[21] Navarro, A., Asenjo, R., Tabik, S. and Cascaval, C.: Analytical modeling of pipeline parallelism, *18th International Conference on Parallel Architectures and Compilation Techniques, PACT '09*, pp.281–290, IEEE (2009).

[22] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V. and Fedorova, A.: The Linux scheduler: a decade of wasted cores, *Proc. 11th European Conference on Computer Systems*, p.1, ACM (2016).

[23] Kerrisk, M.: Sched - overview of CPU scheduling, Linux man-pages project (online), available from ⟨http://man7.org/linux/man-pages/man7/sched.7.html⟩ (accessed 2017-05-31).

[24] Shalloway, A. and Trott, J.R.: *Design patterns explained: A new perspective on object-oriented design, 2/E*, Pearson Education India (2005).

[25] Bonwick, J. et al.: The Slab Allocator: An Object-Caching Kernel Memory Allocator, *USENIX Summer*, Vol.16, Boston, MA, USA (1994).

[26] Rusling, D.A.: The Linux Kernel: Memory Management, Linux Documentation Project (online), available from ⟨http://www.tldp.org/LDP/tlk/mm/memory.html⟩ (accessed 2017-05-31).

[27] Clements, A.T., Kaashoek, M.F. and Zeldovich, N.: Scalable address spaces using RCU balanced trees, *ACM SIGPLAN Notices*, Vol.47, No.4, pp.199–210 (2012).

[28] Linux.com: What's New in Linux 2.6.39: Ding Dong, the Big Kernel Lock is Dead, The Linux Foundation (online), available from ⟨https://www.linux.com/learn/whats-new-linux-2639-ding-dong-big-kernel-lock-dead⟩ (accessed 2017-05-31).

[29] Anderson, T.E.: The performance of spin lock alternatives for shared-money multiprocessors, *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.1, pp.6–16 (1990).

[30] Boyd-Wickizer, S., Kaashoek, M.F., Morris, R. and Zeldovich, N.: Non-scalable locks are dangerous, *Proc. Linux Symposium*, pp.119–130 (2012).

[31] Auslander, M., Edelsohn, D., Krieger, O., Rosenburg, B. and Wisniewski, R.: Enhancement to the MCS lock for increased functionality and improved programmability (2002).

[32] ISO: *ISO/IEC 9899:2011 Information technology — Programming languages — C*, International Organization for Standardization, Geneva, Switzerland (2011).

[33] Sourceware: The Newlib Homepage, Red Hat, Inc. (online), available from ⟨https://sourceware.org/newlib⟩ (accessed 2017-05-31).

[34] GNU Project: The GNU C Library, Free Software Foundation (online), available from ⟨https://www.gnu.org/software/libc/libc.html⟩ (accessed 2017-05-31).

[35] Larson, P.-Å. and Krishnan, M.: Memory allocation for long-running server applications, *ACM SIGPLAN Notices*, Vol.34, No.3, pp.176–185, ACM (1998).

[36] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: A new dynamic memory allocator for real-time systems, *Proc. 16th Euromicro Conference on Real-Time Systems, ECRTS 2004*, pp.79–88, IEEE (2004).

[37] Lea, D. and Gloger, W.: A memory allocator (1996).

[38] Berger, E.D., McKinley, K.S., Blumofe, R.D. and Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications, *ACM Sigplan Notices*, Vol.35, No.11, pp.117–128 (2000).

[39] Lingegowda, V.: Optimization of Data Read/Write in a Parallel Application, Intel (online), available from ⟨https://software.intel.com/en-us/blogs/2013/03/04/optimization-of-data-readwrite-in-a-parallel-application⟩ (accessed 2017-05-31).

[40] Roth, M., Best, M.J., Mustard, C. and Fedorova, A.: Deconstructing the overhead in parallel applications, *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp.59–68, IEEE (2012).

Yixiao Li received his B.E. degree in Software Engineering from East China Normal University in 2012, and the degree of Master of Information Science from Nagoya University in 2015. He is a Ph.D. candidate at the Graduate School of Information Science, Nagoya University. His research interests include real-time operating systems, embedded multi/many-core systems, and software platforms for embedded systems.

Yutaka Matsubara is an Assistant Professor at the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from Nagoya University in 2011. He was a Researcher from 2009 to 2011 at Nagoya university, and was a designated Assistant Professor in 2012. His research interests include real-time operating systems, real-time scheduling theory, and system safety and security for embedded systems. He is a member of IEEE and USENIX.

Hiroaki Takada is a professor at Institutes of Innovation for Future Society, Nagoya University. He is also a professor and the Executive Director of the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from University of Tokyo in 1996. He was a Research Associate at University of Tokyo from 1989 to 1997, and was a Lecturer and then an Associate Professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IEICE, JSSST, and JSAE.