Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems

2012 Linux Symposium

Geunsik Lim
Sungkyungkwan University
Samsung Electronics
leemgs@ece.skku.ac.kr
geunsik.lim@samsung.com

Changwoo Min
Sungkyungkwan University
Samsung Electronics
multics69@ece.skku.ac.kr
changwoo.min@samsung.com

YoungIk Eom
Sungkyungkwan University
yieom@ece.skku.ac.kr

Abstract

Most commercial embedded devices have been deployed with a single processor architecture. The code size and complexity of applications running on embedded devices are rapidly increasing due to the emergence of application business models such as Google Play Store and Apple App Store. As a result, a high-performance multicore CPUs have become a major trend in the embedded market as well as in the personal computer market.

Due to this trend, many device manufacturers have been able to adopt more attractive user interfaces and high-performance applications for better user experiences on the multicore systems.

In this paper, we describe how to improve the real-time performance by reducing the user waiting time on multicore systems that use a partitioned per-CPU run queue scheduling technique. Rather than focusing on naive load-balancing scheme for equally balanced CPU usage, our approach tries to minimize the cost of task migration by considering the importance level of running tasks and to optimize per-CPU utilization on multicore embedded systems.

Consequently, our approach improves the real-time characteristics such as cache efficiency, user responsiveness, and latency. Experimental results under heavy background stress show that our approach reduces the average scheduling latency of an urgent task by 2.3 times.

1 Introduction

Performance improvement by increasing the clock speed of a single CPU results in a power consumption problems [19, 8]. Multicore architecture has been widely used to resolve the power consumption problem as well as to improve performance [24]. Even in embedded systems, the multicore architecture has many advantages over the single-core architecture [17].

Modern operating systems provide multicore aware infrastructure including SMP scheduler, synchronization [16], interrupt load-balancer, affinity facilities [22, 3], CPUSETS [25], and CPU isolation [23, 7]. These functions help running tasks adapt to system characteristics very well by considering CPU utilization.

Due to technological changes in the embedded market, OS-level load-balancing techniques have been highlighted more recently in the multicore based embedded environment to achieve high-performance. As an example, the needs of real-time responsiveness characteristics [1] have increased by adopting multicore architecture to execute CPU-intensive embedded applications within the desired time on embedded products such as a 3D DTV and a smart phone.

In embedded multicore systems, efficient load-balancing of CPU-intensive tasks is very important for achieving higher performance and reducing scheduling latency when many tasks running concurrently. Thus, it can be the competitive advantage and differentiation.

In this paper, we propose a new solution, *operation zone based load-balancer*, to improve the real-time performance [30] on multicore systems. It reduces the user
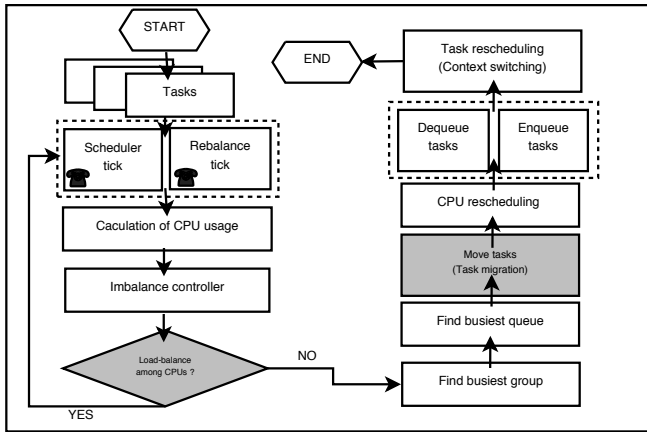
Figure 1: Load-balancing operation on Linux

waiting time by using a partitioned scheduling—or per-CPU run-queue scheduling—technique. Our solution minimizes the cost of task migration [21] by considering the importance level of running tasks and per-CPU utilization rather than focusing on naive CPU load-balancing for balanced CPU usage of tasks.

Finally, we introduce a flexible task migration method according to *load-balancing operation zone*. Our method improves operating system characteristics such as cache efficiency, effective power consumption, user responsiveness, and latency by re-balancing the activities that try to move specific tasks to one of the CPUs on embedded devices. This approach is effective on the multicore-based embedded devices where user responsiveness is especially important from our experience.

2   Load-balancing mechanism on Linux

The current SMP scheduler in Linux kernel periodically executes the load-balancing operation to equally utilize each CPU core whenever load imbalance among CPU cores is detected. Such aggressive load-balancing operations incur unnecessary task migrations even when the CPU cores are not fully utilized, and thus, they incur additional cache invalidation, scheduling latency, and power consumption. If the load sharing of CPUs is not fair, the multicore scheduler [10] makes an effort to solve the system's load imbalance by entering the procedure for load-balancing [11]. Figure 1 shows the overall operational flow when the SMP scheduler [2] performs the load-balancing.

At every timer tick, the SMP scheduler determines whether it needs to start load-balancing [20] or not, based on the number of tasks in the per-CPU run-queue. At first, it calculates the average load of each CPU [12]. If the load imbalance between CPUs is not fair, the load-balancer selects the task with the highest CPU load [13], and then lets the migration thread move the task to the target CPU whose load is relatively low. Before migrating the task, the load-balancer checks whether the task can be instantly moved. If so, it acquires two locks, `busiest->lock` and `this_rq->lock`, for synchronization before moving the task. After the successful task migration, it releases the previously held double-locks [5]. The definitions of the terms in Figure 1 are as follows [10] [18]:

- Rebalance_tick: update the average load of the run-queue.

- Load_balance: inspect the degree of load imbalance of the scheduling domain [27].

- Find_busiest_group: analyze the load of groups within the scheduling domain.

- Find_busiest_queue: search for the busiest CPU within the found group.

- Move_tasks: migrate tasks from the source run-queue to the target run-queue in other CPU.

- Dequeue_tasks: remove tasks from the external run-queue.

- Enqueue_tasks: add tasks into a particular CPU.

- Resched_task: if the priority of moved tasks is higher than that of current running tasks, preempt the current task of a particular CPU.

At every tick, the `scheduler_tick()` function calls `rebalance_tick()` function to adjust the load of the run-queue that is assigned to each CPU. At this time, load-balancer uses `this_cpu` index of local CPU, `this_rq`, `flag`, and `idle` (`SCHED_IDLE`, `NOT_IDLE`) to make a decision. The `rebalance_tick()` function determines the number of tasks that exist in the run-queue. It updates the average load of the run-queue by accessing `nr_running` of the run-queue descriptor and `cpu_load` field for all domains from the default domain to the domain of the upper layer. If the

load imbalance is found, the SMP scheduler starts the procedure to balance the load of the scheduling domain by calling `load_balance()` function.

It is determined by `idle` value in the `sched_domain` descriptor and other parameters how frequently load-balancing happens. If `idle` value is `SCHED_IDLE`, meaning that the run-queue is empty, `rebalance_tick()` function frequently calls `load_balance()` function. On the contrary, if `idle` value is `NOT_IDLE`, the run-queue is not empty, and `rebalance_tick()` function delays calling `load_balance()` function. For example, if the number of running tasks in the run-queue increases, the SMP scheduler inspects whether the load-balancing time [4] of the scheduling domain belonging to physical CPU needs to be changed from 10 milliseconds to 100 milliseconds.

When `load_balance()` function moves tasks from the busiest group to the run-queue of other CPU, it calculates whether Linux can reduce the load imbalance of the scheduling domain. If `load_balance()` function can reduce the load imbalance of the scheduling domain as a result of the calculation, this function gets parameter information like `this_cpu`, `this_rq`, `sd`, and `idle`, and acquires spin-lock called `this_rq->lock` for synchronization. Then, `load_balance()` function returns `sched_group` descriptor address of the busiest group to the caller after analyzing the load of the group in the scheduling domain by calling `find_busiest_group()` function. At this time, `load_balance()` function returns the information of tasks to the caller to move the tasks into the run-queue of local CPU for the load-balancing of scheduling domain.

The kernel moves the selected tasks from the busiest run-queue to `this_rq` of another CPU. After turning on the flag, it wakes up `migration/*` kernel thread. The migration thread scans the hierarchical scheduling domain from the base domain of the busiest run-queue to the top in order to find the most idle CPU. If it finds relatively idle CPU, it moves one of the tasks in the busiest run-queue to the run-queue of relatively idle CPU (calling `move_tasks()` function). If a task migration is completed, kernel releases two previously held spin-locks, `busiest->lock` and `this_rq->lock`, and finally it finishes the task migration.

`dequeue_task()` function removes a particular task in the run-queue of other CPU. Then, `enqueue_task()` function adds a particular task into the run-queue of lo-

cal CPU. At this time, if the priority of the moved task is higher than the current task, the moved task will preempt the current task by calling `resched_task()` function to gain the ownership of CPU scheduling.

As we described above, the goal of the load-balancing is to equally utilize each CPU [9], and the load-balancing is performed after periodically checking whether the load of CPUs is fair. The load-balancing overhead is controlled by adjusting frequency of load-balancing operation, `load_balance()` function, according to the number of running tasks in the run-queue of CPU. However, since it always performs load-balancing whenever a load imbalance is found, there is unnecessary load-balancing which does not help to improve overall system performance.

In multicore embedded systems running many user applications at the same time, load imbalance will occur frequently. In general, more CPU load leads to more frequent task migration, and thus, incurs higher cost. The cost can be broken down into direct, indirect, and latency costs as follows:

1. Direct cost: the load-balancing cost by checking the load imbalance of CPUs for utilization and scalability in the multicore system

2. Indirect cost: cache invalidation and power consumption

   (a) cache invalidation cost by task migration among the CPUs

   (b) power consumption by executing more instructions according to aggressive load-balancing

3. Latency cost: scheduling latency and longer non-preemptible period

   (a) scheduling latency of the low priority task because the migration thread moves a number of tasks to another CPU [29]

   (b) longer non-preemptible period by holding the double-locking for task migration

We propose our *operation zone based load-balancer* in the next section to solve those problems.
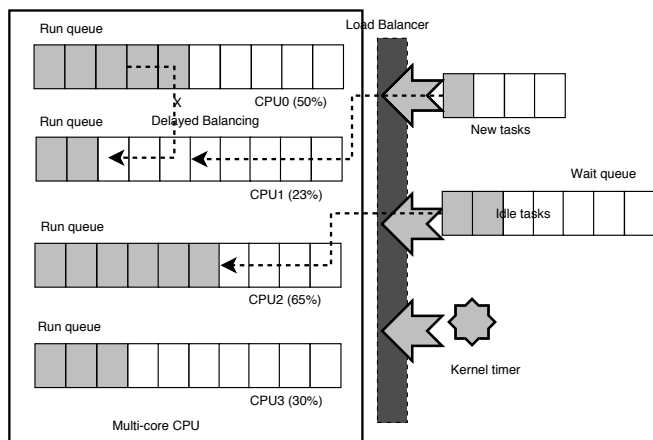
Figure 2: Flexible task migration for low latency

Figure 3: Load-balancing operation zone

3 Operation zone based load-balancer

In this section, we propose a novel load-balancing scheduler called *operation zone based load-balancer* which flexibly migrates tasks for load-balancing based on *load-balancing operation zone* mechanism which is designed to avoid too frequent unnecessary load-balancing. We can minimize the cost of the load-balancing operation on multicore systems while maintaining overall CPU utilization balanced.

The existing load-balancer described in the previous section regularly checks whether load-balancing is needed or not. On the contrary, our approach checks only when the status of tasks can be changed. As illustrated in Figure 2, *operation zone based load-balancer* checks whether the task load-balancing is needed in the following three cases:

- A task is newly created by the scheduler.

- An idle task wakes up for scheduling.

- A running task belongs to the busiest scheduling group.

The key idea of our approach is that it defers load-balancing when the current utilization of each CPU is not seriously imbalanced. By avoiding frequent unnecessary task migration, we can minimize heavy double-lock overhead and reduce power consumption of a battery backed embedded device. In addition, it controls
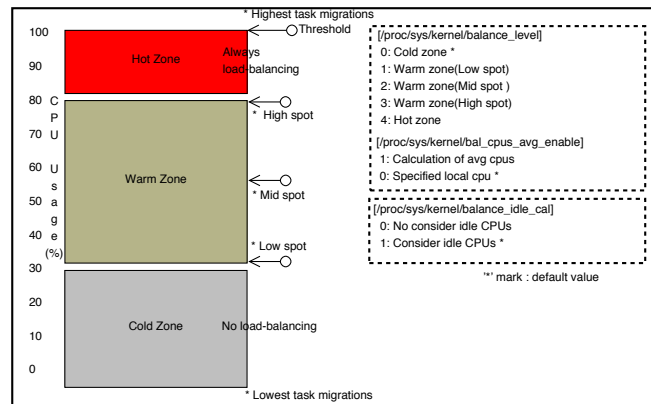
the worst-case scenario: one CPU load exceeds 100% even though other CPUs are not fully utilized. For example, when a task in `idle`, `newidle`, or `noactive` state is rescheduled, we can make the case that does not execute `load_balance()` routine.

3.1 Load-balancing operation zone

Our *operation zone based load-balancer* provides *load-balancing operation zone* policy that can be configured to the needs of the system. As illustrated in Figure 3, it provides three multicore load-balancing policies based on the CPU utilization. The *cold zone* policy loosely performs load-balancing operation; it is adequate when the CPU utilization of most tasks is low.

On the contrary, the *hot zone* policy performs load-balancing operation very actively, and it is proper under high CPU utilization. The *warm zone* policy takes the middle between *cold zone* and *hot zone*.

Load-balancing under the *warm zone* policy is not trivial because CPU utilization in *warm zone* tends to fluctuate continuously. To cope with such fluctuations, *warm zone* is again classified into three spots—high, mid, and low—and our approach adjusts scores based on weighted values to further prevent unnecessary task migration caused by the fluctuation. We provide `/proc` interfaces for a system administrator to configure the policy either statically or dynamically. From our experience, we recommend that a system administrator configures the policy statically because of system complexity.

3.1.1 Cold zone

In a multicore system configured with the *cold zone* policy, our operation zone based load-balancing scheduler does not perform any load-balancing if the CPU utilization is in *cold zone*, 0~30%. Since there is no task migration in *cold zone*, a task can keep using the currently assigned CPU. Kernel performs the load-balancing only when the CPU utilization exceeds *cold zone*.

This policy is adequate where the CPU utilization of a device tends to be low except for some special cases. It also helps to extend battery life in battery backed devices.

3.1.2 Hot zone

Task migration in the *hot zone* policy is opposite to that in the *cold zone* policy. If the CPU utilization is in *hot zone*, 80~100%, kernel starts to perform load-balancing. Otherwise, kernel does not execute the procedure of load-balancing at all.

Under the *hot zone* policy, kernel defers load-balancing until the CPU utilization reaches *hot zone*, and thus, we can avoid many task migrations. This approach brings innovative results in the multicore-based system for the real-time critical system although the system throughput is lost.

3.1.3 Warm zone

In case of the *warm zone* policy, a system administrator chooses one of the following three spots to minimize the costs of the load-balancing operation for tasks whose CPU usage is very active.

- High spot (80%): This spot has the highest CPU usage in the *warm zone* policy. The task of *high spot* cannot go up any more in the *warm zone* policy

- Low spot (30%): This spot has the lowest CPU usage in the *warm zone* policy. The task of *low spot* cannot go down any more in the *warm zone* policy.

- Mid spot (50%): This spot is in between high spot and low spot. The weight-based dynamic score adjustment scheme is used to cope with fluctuations of CPU utilization.
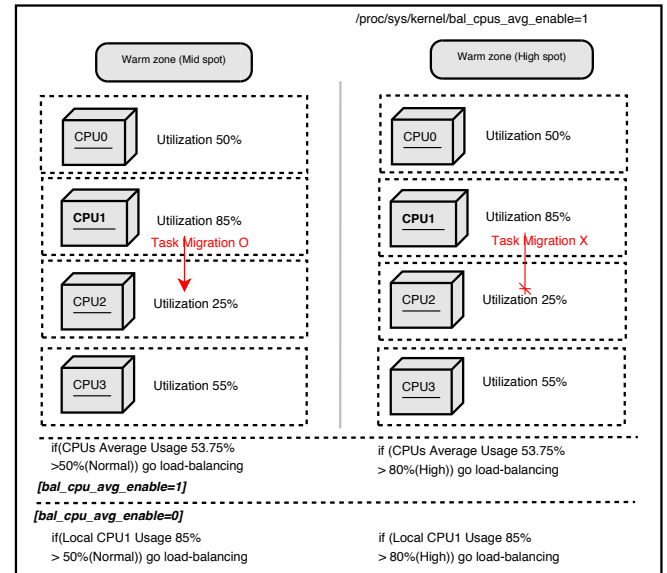

Figure 4: Task migration example in warm zone policy

The spot performs the role of controlling the CPU usage of tasks that can be changed according to weight score. In the *warm zone* policy system, weight-based scores are applied to tasks according to the period of the CPU usage ratio based on *low spot*, *mid spot* and *high spot*. The three spots are detailed for controlling active tasks by users. The CPU usages of tasks have penalty points or bonus points according to the weight scores.

Although the score of task can increase or decrease, these tasks cannot exceed the maximum value, *high spot*, and go below the minimum value, *low spot*. If CPU usage of a task is higher than that of the configured spot in the *warm zone* policy, kernel performs load-balancing through task migration. Otherwise, kernel does not execute any load-balancing operation.

For example, we should consider that the CPU utilization of quad-core systems is 50%, 85%, 25% and 55% respectively from CPU0 to CPU3 as Figure 4. If the system is configured in *mid spot* of the *warm zone* policy, the load-balancer starts operations when the average usage of CPU is over 50%. Kernel moves one of the running tasks of run-queue in CPU1 with the highest utilization to the run-queue of CPU2 with the lowest utilization.

In case of *high spot* and the *warm zone* policy, the load-balancer starts the operations when the average usage of CPU is over 80%. Tasks that are lower than the CPU usage of the *warm zone* area is not migrated into another
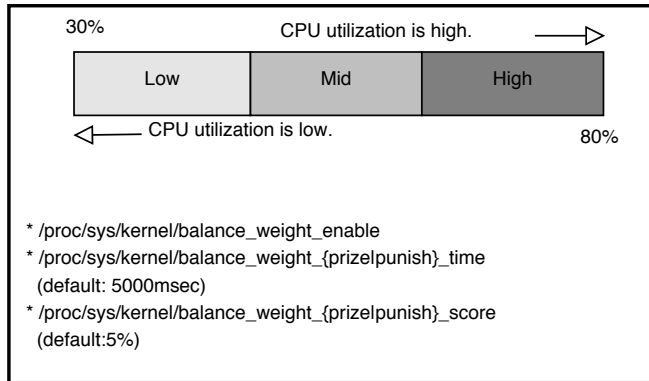
Figure 5: Weight-based score management

CPU according to migration thread. Figure 4 depicts the example of load-balancing operations on the *warm zone* policy.

Figure 5 shows weight-based load score management for the *warm zone* policy system. When the usage period of CPU is longer than the specified time, five seconds by default, kernel manages bonus points and penalty points to give relative scores to the task that utilizes CPU resources continually and actively. Also, kernel operates the load weight-based *warm zone* policy to support the possibility that the task can use the existing CPU continually.

At this time, tasks that reach the level of the *high spot*, stay in the *warm zone* range although the usage period of CPU is very high. Through these methods, kernel keeps the border of the *warm zone* policy without moving a task to the *hot zone* area.

If a task maintains the high value of the usage of CPU more than five seconds as the default policy based on `/proc/sys/kernel/balance_weight_ {prize|punish}_time`, kernel gives the task CPU usage score of -5 which means that CPU utilization is lower. At this point, the CPU usage information of the five seconds period is calculated by the scheduling element of a task via *proc file system*. We assigned the five seconds by default via our experimental experience. This value can be changed by using `/proc/sys/kernel/balance_weight_{prize| punish}_time` by the system administrator to support various embedded devices.

In contrast, if a task consumes the CPU usage of a spot shorter than five seconds, kernel gives the task CPU us-

age score of +5 which means that CPU utilization is higher. The task CPU usage score of +5 elevates the load-balancing possibility of tasks. Conversely, the task CPU usage score of -5 aims to bring down the load-balancing possibility of tasks.

The value of the *warm zone* policy is static, which means it is determined by a system administrator without dynamic adjustment. Therefore, we need to identify active tasks that consume the usage of CPUs dynamically. The load weight-based score management method calculates a task's usage in order that kernel can consider the characteristics of these tasks. This mechanism helps the multicore-based system manage the efficient load-balancing operation for tasks that have either high CPU usage or low CPU usage.

3.2 Calculating CPU utilization

In our approach, the CPU utilization plays an important role in determining to perform load-balancing. In measuring CPU utilization, our approach provides two ways: calculating CPU utilization for each CPU and averaging CPU utilization of all CPUs. A system administrator also can change behaviors through `proc` interface, `/proc/sys/kernel/balance_cpus_ avg_enable`. By default, kernel executes task migration depending on the usage ratio of each CPU.

If a system administrator selects `/proc/system/ kernel/balance_cpus_avg_enable=1` parameter for their system, kernel executes task migration depending on the average usage of CPUs.

The method to compare load-balancing by using the average usage of CPUs, helps to affinitize the existing CPU as efficiently as possible for some systems. The system needs the characteristics of CPU affinity [14] although the usage of a specific CPU is higher than the value of the *warm zone* policy, e.g. CPU-intensive single-threaded application in the most idle systems.

4 Evaluation

4.1 Evaluation scenario

Figure 6 shows our evaluation scenario to measure the real-time characteristics of running tasks in multicore based embedded systems. In this experiment, we measured how scheduling latency of an urgent task would
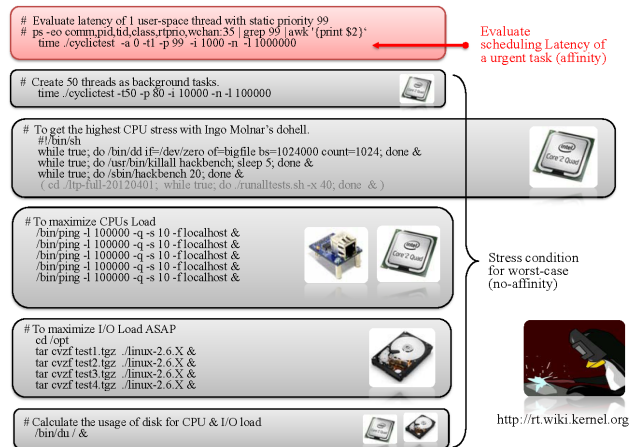
Figure 6: Evaluation scenario to measure scheduling latency

Figure 7: Comparison of scheduling latency distribution

be reduced under very high CPU load, network stress, and disk I/O.

To measure scheduling latency, we used *cyclictest* utility of *rt-test* package [28] which is mainly used to measure real-time characteristics of *Redhat Enterprise Linux (RHEL)* and real-time Linux. All experiments are performed in `Linux2.6.32` on `IntelQuadcoreQ9400`.

4.2 Experimental result

In Figure 7, we compared the scheduling latency distribution between the existing approach (before) and our proposed approach (after).

Our approach is configured to use *warm zone - high spot* policy. Under heavy background stress reaching to the worst load to the Quad-core system, we measured the scheduling latency of our test thread which repeatedly sleeps and wakes up. Our test thread is pinned to a particular CPU core by setting CPU affinity [15] and is configured as the FIFO policy with priority 99 to gain the best priority.

In the figure, X-axis is the time from the test start, and Y-axis is the scheduling latency in microseconds from when it tries to wake up for rescheduling after a specified sleep time. As Figure 7 shows, the scheduling latency of our test thread is reduced more than two times: from 72 microseconds to 31 microseconds on average.

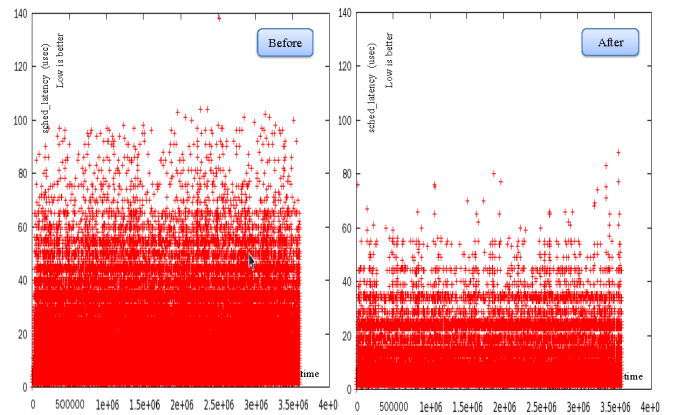In order to further understand why our approach reduces scheduling latency more than two times, we traced the caller/callee relationship of all kernel function during the experiment by using Linux internal function tracer, *ftrace* [26].

The analysis of the collected traces confirms three: first, the scheduling latency of a task can be delayed when migration of other task happens. Second, when task migration happens, non-preemptible periods are increased for acquiring double-locking. Finally, our approach can reduce average scheduling latency of tasks by effectively removing vital costs caused by the load-balancing of the multicore system.

In summary, since the migration thread is a real-time task with the highest priority, acquiring double-locking and performing task migration, the scheduling of the other tasks can be delayed. Since load imbalance frequently happens under a heavily loaded system with many concurrent tasks, the existing very fair load balancer incurs large overhead, and our approach can reduce such overhead effectively.

Our *operation zone based load-balancer* performs load-balancing based on CPU usage with lower overhead while avoiding overloading to a particular CPU that can increase scheduling latency. Moreover, since our approach is implemented only in the operating system, no modifications of user applications are required.

5 Further work

In this paper, we proposed an operation zone based load-balancing mechanism which reduces scheduling latency. Even though it reduces scheduling latency, it does not guarantee deadline for real-time systems where

the worst case is most critical. In order to extend our approach to the real-time tasks, we are considering a hybrid approach with the physical CPU shielding technique [6] which dedicates a CPU core for a real-time task. We expect that such approach can improve real-time characteristics of a CPU intensive real-time task.

Another important aspect especially in embedded systems is power consumption. In order to keep longer battery life, embedded devices dynamically turn on and off CPU cores. To further reduce power consumption, we will extend our load-balancing mechanism considering CPU on-line and off-line status.

We experimented with scheduling latency to enhance the user responsiveness on the multicore-based embedded system in this paper. We have to evaluate various scenarios such as direct cost, indirect cost, and latency cost to use our load-balancer as a next generation SMP scheduler.

6   Conclusions

We proposed a novel *operation zone based load-balancing technique* for multicore embedded systems. It minimized task scheduling latency induced by the load-balancing operation. Our experimental results using the *cyclictest* utility [28] showed that it reduced scheduling latency and accordingly, users' waiting time.

Since our approach is purely kernel-level, there is no need to modify user-space libraries and applications. Although the vanilla Linux kernel makes every effort to keep the CPU usage among cores equal, our proposed *operation zone based load-balancer* schedules tasks by considering the CPU usage level to settle the load imbalance.

Our design reduces the non-preemptible intervals that require double-locking for task migration among the CPUs, and the minimized non-preemptible intervals contribute to improving the software real-time characteristics of tasks on the multicore embedded systems.

Our scheduler determines task migration in a flexible way based on the *load-balancing operation zone*. It limits the excess of 100% usage of a particular CPU and suppresses the task migration to reduce high overhead for task migration, cache invalidation, and high synchronization cost. It reduces power consumption and scheduling latency in multicore embedded systems, and

thus, we expect that customers can use devices more interactively for longer time.

7   Acknowledgments

References

[1] J.H. Anderson. Real-time scheduling on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium*, 2006.

[2] ARM Information Center. Implementing DMA on ARM SMP System. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0228a/index.html.

[3] M Astley. Migration policies for multicore fair-share schedulingd choffnes. In *ACM SIGOPS Operating Systems*, 2008.

[4] Ali R. Behrooz A. Shirazi, Krishna M. Kavi. Scheduling and load balancing in parallel and distributed systems. In *IEEE Computer Society Press Los Alamitos*, 1995.

[5] Stefano Bertozzi. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *In Proceeding DATE '06 Proceedings of the conference on Design, automation and test in Europe*, 2006.

[6] S Brosky. Shielded processors: Guaranteeing sub-millisecond response in standard linux. In *Parallel and Distributed Processing*, 2003.

[7] S Brosky. Shielded cpus: real-time performance in standard linux. In *Linux Journal*, 2004.

[8] Jeonghwan Choi. Thermal-aware task scheduling at the system software level. In *In Proceeding ISLPED '07 Proceedings of the 2007 international symposium on Low power electronics and design*, 2007.

[9] Slo-Li Chu. Adjustable process scheduling mechanism for a multiprocessor embedded system. In *6th WSEAS international conference on applied computer science*, 2006.

[10] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd Edition. O'Reilly Media.

[11] Mor Harchol-Balter. Exploiting process lifetime distributions for dynamic load balancing. In *ACM Transactions on Computer Systems (TOCS)*, volume 15, 1997.

[12] Toshio Hirosaw. Load balancing control method for a loosely coupled multi-processor system and a device for realizing same. In *Hitachi, Ltd., Tokyo, Japan, Patent No. 4748558*, May 1986.

[13] Steven Hofmeyr. Load balancing on speed. In *PPoPP '10 Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[14] Vahid Kazempour. Performance implications of cache affinity on multicore processors. In *EURO-PAR*, 2008.

[15] KD Abramson. Affinity scheduling of processes on symmetric multiprocessing systems. http://www.google.com/patents/US5506987.

[16] Knauerhase. Using os observations to improve performance in multicore systems. In *Micro IEEE*, May 2008.

[17] Markus Levy. Embedded multicore processors and systems. In *Micro IEEE*, May 2009.

[18] Linus Toavalds. Linux Kernel. http://www.kernel.org.

[19] Andreas Merkel. Memory-aware scheduling for energy efficiency on multicore processors. In *HotPower'08 Proceedings of the 2008 conference on Power aware computing and systems*, 2008.

[20] Nikhil Rao, Google. Improve load balancing when tasks have large weight differential. http://lwn.net/Articles/409860/.

[21] Pittau. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Embedded Systems for Real-Time Multimedia*, 2007.

[22] Robert A Alfieri. Apparatus and method for improved CPU affinity in a multiprocessor system. http://www.google.com/patents/US5745778.

[23] H PÃűtzl S Soltesz. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS*, 2007.

[24] Suresh. Siddha. Ottawa linux symposium (ols). In *Chip Multi Processing (CMP) aware Linux Kernel Scheduler*, 2005.

[25] Simon Derr, Paul Menage. CPUSETS. http://http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt.

[26] Steven Rostedt. Ftrace (Kernel function tracer). http://people.redhat.com/srostedt.

[27] Suresh Siddha. sched: new sched domain for representing multicore. http://lwn.net/Articles/169277/.

[28] Thomas Gleixner, Clark williams. rt-tests (Real-time test utility). http://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git.

[29] V Yodaiken. A real-time linux. In *Proceedings of the Linux Applications*, 1997.

[30] Yuanfang Zhang. Real-time performance and middleware on multicore linux platforms. In *Washington University*, 2008.