Real-Time Scheduling Parallel Tasks on Multicore Platforms

Zhenyang Lei, Xiangdong Lei* and Jun Long
School of Computer Science and Engineering, Central South University, Changsha, People’s Republic of China
*Corresponding Author: leixiangdong@csu.edu.cn

Abstract. Shared resources on the multicore chip, such as main memory, are increasingly becoming a point of contention. Traditional real-time task scheduling strategies focus on solely on the CPU, and do not take in account memory access and cache effects. In this paper, we propose real-time parallel scheduling (PRTTS) strategy on multicore platforms. Each task is represented as a directed acyclic graph (DAG). Tasks priorities are assigned according to task periods. In PRTTS scheduling strategy priorities of tasks which access memory are promoted over priorities of tasks not accessing memory. Tasks which read/write data in cache dynamically have their priority increased above all tasks. The results of simulation experiment show that proposed new scheduling strategy offers better performance in terms of core utilization and schedulability rate of tasks.

1. Introduction
Shared resources on multicore chips, such as main memory, are increasingly becoming a point of contention [1][2]. With the increase of the number of cores, more master components can simultaneously access main memory, and the worst-case task execution time can grow linearly with the number of cores in the system. Traditional real-time task scheduling strategies focus solely on the CPU. The traditional real-time task scheduling strategies may work poorly, especially when memory load is high, due to an increased worst-case stall time when accessing main memory [3]. Classic multicore scheduling strategies, therefore, are no longer sufficient to guarantee schedulability of tasks.

Many researchers have studied the task scheduling on multicore platforms. Lee et al. [4] proposed limited preemptive scheduling strategy on multicore systems. Rosen et al. [5] a method to obtain efficient TDMA arbitration strategy in which tasks are statically partitioned to cores and each core is allowed to access memory only during its granted TDMA slot. Gang et al. [6] proposed global real-time memory centric scheduling strategy for memory-intensive task sets. Gang et al. [7] proposed a memory centric scheduling strategy for hard real-time tasks. However, these scheduling strategies do not take into account the effect of cache in multicore systems.

Because of cache effects, real-time task scheduling is challenging. Migration of locked cache lines for task migration under a global preemptive scheduling method is proposed [8]. Mancuso et al. [9] proposed a cache management framework that combines page coloring and cache-lockdown to minimize cache interference in multicore platforms. The interference in memory is an important problem in multicore platform. One direction is to minimize memory interference by proposing a new hardware mechanism, but it needs to be modified on hardware [10]. The other direction is based on the software method, which coordinates the main memory access to minimize interference.

2. Parallel Task Model
Suppose there are n parallel tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in the system, which are divided into w task sets, and the tasks in the task sets share data. Task $\tau_i$ (1 $\leq$ i $\leq$ n) is related to (Ti, Ri, Di, Pi). In this quadruple
(Ti, Ri, Di, Pi), Ti represents the period of task τi, Ri represents the worst-case execution time (WCET) of task τi, Di represents the relative deadline of task τi, and Pi represents the priority of task τi. Each task is assigned a priority. The priorities of tasks are assigned according to task periods, that is, tasks with shorter cycles are given higher system priority.

Each task is composed of three phases. The first phase is read memory stage, the second phase is execution phase and the third phase is write memory phase.

Real-time synchronization and locking protocols have been developed to ensure mutual exclusion and bounded blocking time in a multicore environment. The tasks use locks and critical sections to protect access. After a task acquires lock, the task reads or writes, and immediately releases lock. Therefore, if a task acquires lock in the first phase, the task will read sharing data and immediately releases lock. If a task acquires lock in the third phase, the task will write its results to sharing data and release immediately lock.

There is a dependency constraint between tasks in the task set, which specifies whether some tasks need to be ahead of others. If the output (write data) of task τi is to be used as the input (read data) of task τj, then task τj is constrained to execute τi first. The precedence constraint can be expressed as directed acyclic graph (DAG) Gk=(Vk, Ek) (1≤k≤w), where node vi∈Vk represents task, Ek=Vk×Vk, and edge ej∈Ek represents dependency between nodes. τi→τj indicates that the task τj is constrained to be executed first. d(τi) represents the set of priority tasks of task τi, that is to say, d(τi) represents the tasks that must be completed before task τi starts. Parallel tasks τ={τ1, τ2, ... , τn} can also be expressed as Γ=(G1, G2, ..., Gw).

In this paper, we present new parallel real-time tasks scheduling (PRTTS) strategy for parallel task sets based on the global scheduling algorithm in multicore platforms. We consider global real-time task scheduling. The global scheduler maintains the task pool p (τ) in which tasks are ready to be executed, that is, there are no dependency between these tasks, which can run on any core. When the task is in the execution phase, the core is occupied by this task. In addition, each task must be executed serially and cannot run in parallel on multiple cores. The system consists of a multicore chip Ci (0≤i≤c-1) with c identical cores, C={C0, C2, ..., Cc-1}, with m memory bank Mj (0≤j≤m-1), M={M0, M0, ..., Mm-1}.

3. Scheduling Strategy

The proposed scheduling strategy consists of two level schedulers: the first level scheduler schedules ready real-time tasks in the task pool to cores, and the second level scheduler schedules real-time tasks on cores. More than m cores are not allowed to access main memory at the same time. The time to access main memory does not depend on the number of other cores accessing main memory at the same time. If more than m tasks are ready to access main memory, the lower priority tasks are blocked. Dynamically increase the priority of tasks to access memory, higher than all tasks that do not access memory. When the data read by the task is in the cache, the priority of the task is raised to the highest priority, and the task is scheduled immediately to preempt the core of the task without access to the memory. Because the data is already in the cache, these tasks do not need to be read to the memory, but to the cache. Since all tasks in the read memory phase have higher priority than those in the execution phase, if the number of tasks in the read main memory phase is less than m, the tasks in the read memory phase immediately occupy the core of the tasks in the execution phase. After accessing memory, the priority of these tasks is restored to the original priority and these tasks are pended, the preempted task continues to run on the core. Because the scheduled task has completed the memory stage, as long as the core is idle, the task can run immediately. Therefore, the proposed scheduling strategy improves memory utilization, core utilization and task response time.

PRTTS scheduling strategy consists of two levels, the first level scheduling will schedule the ready parallel real-time tasks in the task pool to the processor cores, and the second level scheduling will schedule the parallel real-time tasks on the cores. The first level of scheduling is to schedule the ready parallel real-time tasks in the task pool to the processor core. The algorithm is described as follows:
Algorithm 1 Scheduling Parallel Real-Time Tasks to Cores

**Input:** Parallel Task \( \tau = \{ \tau_1, \tau_2, \ldots, \tau_n \} \),
- Parallel Task Set \( \Gamma = (G_1, G_2, \ldots, G_w) \)
- Dependent Set: \( \zeta = \{ d(\tau_1), d(\tau_2), \ldots, d(\tau_n) \} \)
- Multicores: \( C = \{ C_1, C_2, \ldots, C_c \} \)
- Memory Bank: \( M = \{ M_1, M_2, \ldots, M_m \} \)

1. begin
2. for \( i = 1 \) to \( n \) do
3.   if \( d(\tau_i) = \emptyset \) then
4.     \( p(\tau) \leftarrow \tau_i \)
5.   end if
6. end for
7. sort(\( p(\tau) \)) in priority;
8. for \( i = 1 \) to \( c \) do
9.   if \( \exists C_i \text{ is idle} \) then
10.      Assign \( C_i \) to \( \tau_j \);
11.      \( p(\tau) \leftarrow p(\tau) - \tau_j \);
12. end if
13. end for
14. while \( p(\tau) \neq \emptyset \) do
15.   if \( \exists \tau_j \text{ in } p(\tau) \text{ reads data } x \text{ and } \text{x is in cache and valid} \) then
16.     Increase the priority of \( \tau_j \) to the highest priority;
17.     \( p(\tau) \leftarrow p(\tau) - \tau_j \);
18. end if
19. else if \( \exists M_k \text{ is not be accessed} \) then
20.     Assign \( C_i \) on which the task running does not access the memory to \( \tau_j \);
21.     Increase the priority of \( \tau_j \) to the highest priority;
22.     \( p(\tau) \leftarrow p(\tau) - \tau_j \);
23. end if
24. end while
25. sort(\( p(\tau) \)) in priority;
26. end

In this algorithm, tasks that are not constrained (can be executed immediately) are first put into task pool \( p(\tau) \), and tasks in task pool are sorted by priority, and tasks of high priority are scheduled to cores. If memory bandwidth is available, the priorities of tasks to access main memory will be increased dynamically, which is higher than all tasks that do not access memory. When the data read by the task is in cache, the priority of the task is raised to the highest priority, and the task is scheduled immediately to preempt the core of task without access to main memory.

The second level of scheduling will schedule parallel real-time tasks on the core. The algorithm is described as follows:
Algorithm 2 Scheduling Parallel Real-Time Tasks on Cores

1. begin
2. if there is new \( \tau_j \) task to arrive then
3. if a task \( \tau_k \) is running not accessing memory then
4. task \( \tau_j \) preempts running task \( \tau_k \);
5. if \( (\tau_j \text{ reads data } x) \)
6. if \( (x \text{ is in cache of some core and valid}) \) then
7. task \( \tau_j \) read \( x \) from cache;
8. else
9. task \( \tau_j \) read \( x \) from memory;
10. end if
11. end if
12. Pend task \( \tau_j \);
13. task \( \tau_k \) continues to run on the core;
14. end if
15. end if
16. if task \( \tau_i \) wants to access memory then
17. if \( \exists M_l \text{ can be accessed} \) then
18. task \( \tau_i \) accesses memory;
19. else
20. Pend task \( \tau_i \);
21. Schedule other task pending on the core;
22. end if
23. end if
24. if task \( \tau_i \) finishes then
25. for \( j = 1 \) to \( \tau_i \text{.length} \) do
26. if \( \tau_i \in d(\tau_j) \) then
27. \( d(\tau_j) \leftarrow d(\tau_j) - \tau_i \);
28. end if
29. end for
30. end if
31. end

When a new task arrives, if the running task does not access the memory, the running task immediately is suspended and gives up the core to the new task. If the data to be accessed by the new task is in cache, read the data from cache; otherwise, the new task reads the data from main memory. The new task is suspended after reading data, and the original suspended task continues to run. After task \( \tau_i \) is completed, if \( \tau_i \rightarrow \tau_j \), \( \tau_i \) shall be removed from dependency set of \( \tau_j \).

4. Performance Evaluation

The simulation experiments are aimed at studying the performance of the proposed PRTTS scheduling strategy. The major performance is the schedulability rate of tasks. Other performance metrics include core utilization. Task schedulability is the sum of the completion rate of each task, that is, the number of completed tasks divided by the total number of tasks. The core utilization is defined as the sum of each task’s core utilization, which is computed as the task’s computation time (including all memory and execution phases) divided by the task’s period and the number of cores.

The experimental parameters set the number of cores \( c \) to be 2 to 16, the number of memory bank \( m \) to 2, the number of task sets \( w \) to 9000, and the number of tasks \( u \) to be 2 to 16 for each task set.

For the purpose of compared analysis, global earliest deadline first (GEDF) scheduling strategy [11] is selected for compared analysis with the proposed PRTTS scheduling strategy. We use a similar DAG task set generation algorithm that is used in [12] which generates a series of parallel tasks. In order to evaluate the performance of various scheduling strategies, we develop a simulation system written in C++.
Figure 1 shows the relationship between the schedulability rate of tasks and the number of cores under different scheduling strategies when memory bank $m=2$. It can be seen that the schedulability of tasks in PRTTS scheduling strategy is higher than that of GEDF scheduling strategy. Because PRTTS scheduling strategy considers the effect of cache and improves the utilization of memory and core utilization, the schedulability rate of tasks.

Figure 2 shows the relationship between the schedulability rate of tasks and the number of cores when memory bank $m=4$. It can be seen that the schedulability of tasks in PRTTS and GEDF scheduling strategy will increase with the increase of the number of memory banks. In the real-time system, the memory bandwidth has a great influence. The PRTTS scheduling strategy fully considers the memory bandwidth and prevents the memory interference caused by cores accessing to the memory.

Figure 3 shows the relationship between the schedulability rate of task sets and the number of cores when memory bank $m=2$ and the task number $u=2$ to 8. As the number of cores increases, the schedulability rate of task sets increase in PRTTS and GEDF scheduling strategies. It can be seen that PRTTS scheduling strategy has higher schedulability rate of task sets than GEDF scheduling strategy.
PRTTS scheduling strategy takes into account the effect of cache. The potential priority inversion of GEDF scheduling strategy will lead to the failure of task set scheduling.

Figure 4 shows the relationship between the schedulability rate of task sets and the number of cores when memory bank \( m = 4 \) and the task number \( u = 2 \text{--} 16 \). It can be seen that the advantage of PRTTS scheduling strategy to GEDF scheduling strategy is expanded in the schedulability rate of task sets, which shows that PRTTS scheduling strategy has better performance in the schedulability rate of task sets.

5. Conclusion

In this paper, we propose PRTTS scheduling strategy for parallel real-time tasks. In PRTTS scheduling strategy priorities of tasks which access memory are promoted over priorities of tasks not accessing memory. Tasks which read/write data in cache dynamically have their priority increased above all tasks. The results of simulation experiment show that proposed new scheduling strategy offers better performance in terms of core utilization and schedulability rate of tasks.

6. References

[1] Lei Xiangdong, Zhao Yuelong, Yuan Xiaoli, “Concurrency control in mobile distributed real-time database systems,” Journal of Parallel and Distributed Computing, vol. 69, no.10, 2009, pp. 866-876.

[2] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo, “Schedulability analysis of conditional parallel task graphs in multicore systems,” IEEE Transactions on Computers, vol. 66, no. 2, 2017, pp. 339-353.

[3] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha, “Memory bandwidth management for efficient performance isolation in multi-core platforms,” IEEE Transactions on Computers, vol. 65, no. 2, 2016, pp. 562-576.

[4] J. Lee, K. G. Shin, “Cont rolling preemption for better schedulability in multi-core systems,” in Proc. IEEE 33rd Real-Time Syst. Symp., 2012, pp. 29-38.

[5] J. Rosen, A. Andrei, P. Eles, and Z. Peng, “Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip,” in Proc. 28th IEEE Int. Real-Time Syst. Symp., 2007, pp. 49-60.

[6] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo, “Global Real-Time Memory-centric scheduling for multicore systems,” IEEE Transactions on Computers, vol. 65, no. 9, 2016, pp. 2739-2751.

[7] Gang Yao, R. Pellizzoni, R. Bak S., “Memory centric scheduling for multicore hard real-time systems,” Real-Time System, vol. 48, no. 16, 2012, pp. 681-715.

[8] A. Sarkar, F. Mueller, and H. Ramaprasad, “Predictable task migration for locked caches in multi-core systems,” in Proc. SIGPLAN/SIGBED Conf. Languages, Compilers Tools Embedded Syst., 2011, pp. 131-140.

[9] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multicore architectures,” in Proc. 19th IEEE Real-Time Embedded Technol. Appl. Symp., 2013, pp. 45-54.

[10] B. Akesson, K. Goossens, and M. Ringhofer, “Predator: A predictable SDRAM memory controller,” in Proc. 5th IEEE/ACM Int. Conf. Hardware/Softw. Codes. Syst. Synthesis, 2007, pp. 251-256.

[11] Hoon Sung Chwa, Jinkyu Lee, Jiyeon Lee, Kiew-My Phan, Arvind Easwaran, and Insik Shin, “Global EDF schedulability analysis for parallel tasks on multi-core platforms,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, 2017, pp. 1331-1344.

[12] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, “Parallel Real-Time Scheduling of DAGs,” IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 12, 2014, pp. 3242-3252.