Global Scheduling Real-Time 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
Email: leixiangdong@csu.edu.cn

Abstract. Traditional real-time task scheduling strategies only consider cores scheduling, and do not take into account memory access and cache effects on multicore platforms. In this paper, we propose real-time parallel scheduling (PRTTS) strategy on multicore platforms. Each task is represented as a directed acyclic graph (DAG). The priorities of tasks are assigned according to task periods (rate monotonic, RM). In PRTTS scheduling strategy, priorities of tasks that access memory are promoted over priorities of tasks not accessing memory. Tasks accessing the data in the cache are dynamically raised to highest priority. 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

Many parallel applications, such as parallel transaction processing, parallel sorting, parallel matrix multiplication and parallel fast Fourier transform, are used in real-time systems [1]. On multicore platforms, each core competes for shared resources, such as main memory [2, 3]. 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.

In this paper, we present new parallel real-time tasks scheduling (PRTTS) strategy for parallel task on multicore platforms. Each task set is represented as a directed acyclic graph (DAG). Tasks priorities are assigned according to task periods (rate monotonic, RM). In PRTTS scheduling strategy priorities of tasks that access memory are promoted over priorities of tasks not accessing memory. Tasks accessing the data in the cache are dynamically raised to highest priority.

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] proposed a TDMA arbitration method, in which each core is allowed to access memory only during its specified 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. A lock cache line migration method for task migration under global preemption scheduling method is proposed [8]. To reduce cache interference on multi-core platforms, Mancuso et al. [9] proposed a cache management framework by using page coloring method. In order to minimize memory interference, there are two research directions. One research direction is to propose a new hardware mechanism, but it needs to be
modified on the hardware [10]. Another research direction is a software based approach, which coordinates access to main memory 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 \((T_i, R_i, D_i, P_i)\). In this quadruple \((T_i, R_i, D_i, P_i)\), \(T_i\) represents the period of task \( \tau_i \), \(R_i\) represents the worst-case execution time (WCET) of task \( \tau_i \), \(D_i\) represents the relative deadline of task \( \tau_i \), and \(P_i\) represents the priority of task \( \tau_i \). 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.

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 \( \tau_i \) is to be used as the input (read data) of task \( \tau_n \), then task \( \tau_i \) is constrained to execute \( \tau_n \) first. The precedence constraint can be expressed as directed acyclic graph (DAG) \( G_i=(V_i, E_i) \) \((1 \leq k \leq w)\), where node \( v_i \in V_k \) represents task, \( E_i=V_i \times V_i \), and edge \( e \in E_i \) represents dependency between nodes. \( \tau_i \rightarrow \tau_j \) indicates that the task \( \tau_i \) is constrained to be executed first. \( d(\tau_i) \) represents the set of priority tasks of task \( \tau_i \), that is to say, \( d(\tau_i) \) represents the tasks that must be completed before task \( \tau_i \) starts. Parallel tasks \( \tau = \{\tau_1, \tau_2, \ldots, \tau_n\} \) can also be expressed as \( \Gamma=(G_1, G_2, \ldots, G_w) \). The system consists of a multicore chip \( C_i \) \((0 \leq i \leq c-1)\) with \( c \) identical cores, \( C=\{C_0, C_1, \ldots, C_c\} \), with \( m \) memory bank \( M_j \) \((0 \leq j \leq m-1)\), \( M=\{M_0, M_0, \ldots, M_m\} \).

3. Scheduling Strategy

The proposed PRTTS 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. 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.

In order to better explain the proposed scheduling strategy, a specific task set is given below. The traditional task priority based scheduling strategy and the proposed scheduling strategy are respectively used to schedule this task set. Suppose the system has \( c=3 \) cores \( \{C_0, C_1, C_2\} \) and \( m=1 \) memory bank. There are six parallel tasks \( p(\tau)=\{\tau_1, \tau_2, \ldots, \tau_6\} \) in the system task pool. The task \( \tau_1 \) shares data \( x \) with \( \tau_2 \) and \( \tau_3 \). Task \( \tau_1 \) first acquires lock, reads data \( x \), and releases lock. Then task \( \tau_2 \) acquires lock, reads data \( x \), and releases lock. Finally, task \( \tau_3 \) acquires lock, reads data \( x \), and releases lock. These tasks arrive at time \( t=0 \). Table 1 lists read data, write data and execution time of six tasks.

| Task | Read Data | Write Data | Execution Time |
|------|-----------|------------|----------------|
| \( \tau_1 \) | \( x \) | | 3 |
| \( \tau_2 \) | \( x \) | | 4 |
| \( \tau_3 \) | \( x \) | | 5 |
| \( \tau_4 \) | | \( y \) | 6 |
| \( \tau_5 \) | | \( y \) | 7 |
| \( \tau_6 \) | | \( z \) | 8 |

Figure 1 shows the scheduling of task \( p(\tau)=[\tau_1, \tau_2, \ldots, \tau_6] \) based on the traditional scheduling strategy of task priority. When time \( t=0 \), tasks \( \tau_1, \tau_2 \) and \( \tau_3 \) are scheduled to core \( C_0, C_1 \) and \( C_2 \), respectively. When time \( t=5 \), \( \tau_1 \) accesses the memory, and \( \tau_2 \) and \( \tau_3 \) accesses main memory at time \( t=6 \) and \( t=7 \), respectively. When \( t=6 \), task \( \tau_1 \) has finished on core \( C_0 \). At this time, task \( \tau_4 \) is scheduled to
core $C_0$, task $\tau_4$ can't run immediately, so it must be delayed to time $t=8$ in order to access main memory. Task $\tau_5$ shares data $x$ with $\tau_1$ and $\tau_2$, but when $\tau_5$ is scheduled, data $x$ has failed or been replaced in the cache and has to be read from main memory. Similarly, although task $\tau_6$ is scheduled to core $C_2$ at time $t=7$, it has to be delayed to time $t=10$ to access main memory. From figure 1, it can be found that these six tasks need 16 units of time to complete.

### Table 1. The read data, write data and execution time of six tasks

| Task | read data | Execution phase | write data |
|------|-----------|----------------|------------|
| $\tau_1$ | read $x$ | 4 units of time | write $y$ |
| $\tau_2$ | read $x$ | 3 units of time | write $z$ |
| $\tau_3$ | read $s$ | 2 units of time | write $t$ |
| $\tau_4$ | read $w$ | 1 unit of time | write $w$ |
| $\tau_5$ | read $x$ | 4 units of time | write $r$ |
| $\tau_6$ | read $a$ | 3 units of time | write $b$ |

Figure 2 shows the proposed strategy scheduling for task $p(\tau)=$\{$\tau_1, \tau_2, \ldots, \tau_6$\}. When time $t=0$, tasks $\tau_1$, $\tau_2$ and $\tau_3$ are scheduled to core $C_0$, $C_1$ and $C_2$ respectively. At time $t=1$, since data $x$ to be read by $\tau_2$ is already in cache, data can be read from the cache. At the same time, $\tau_3$ can access main memory when time $t=1$. While the data to be read by task $\tau_5$ has been in cache, the priority of $\tau_5$ is raised to the highest priority, $\tau_5$ is scheduled immediately and preempt core $C_1$ of $\tau_2$. At time $t=2$, $\tau_5$ reads data $x$ from cache, at time $t=3$, the priority of $\tau_5$ is restored to the original priority, $\tau_5$ is suspended, and $\tau_2$ continues to run. Since tasks $\tau_1$ and $\tau_6$ need to access main memory, the priorities of $\tau_4$ and $\tau_6$ are raised to all tasks no accessing to main memory, and $\tau_4$ and $\tau_6$ are scheduled to core $C_0$ and $C_2$, respectively. When time $t=2$, $\tau_4$ accesses main memory, then the priority of $\tau_4$ is restored to the original priority, and is suspended, and $\tau_1$ continues to run. When time $t=3$, $\tau_6$ accesses main memory, then the priority of $\tau_6$ is restored to the original priority, and is suspended, and $\tau_3$ continues to run. It can be seen from figure 2 that these six tasks only need 11 units of time to be completed. PRTTS scheduling strategy improves the parallelism and reduces the response time of task set.

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.

![Figure 1. The traditional scheduling strategy based on task priority for tasks $p(\tau)=$\{$\tau_1, \tau_2, \ldots, \tau_6$\}.](image-url)
Figure 2. PRTTS scheduling strategy for tasks \( \rho(\tau)=\{\tau_1, \tau_2, \ldots, \tau_6\} \).

4. The Necessary Condition for Schedulability of Task

Tasks use locks and critical parts to protect access to shared data. If a task needs to access shared data in the first and third phases, the execution time \( ET_i \) of the task is as follows:

\[
ET_i = \text{Loc} + R_i + \text{Loc} = 2 \text{Loc} + R_i
\]

where, \( \text{Loc} \) and \( R_i \) are time of acquiring the lock and time of normal execution, respectively.

Give a \( G_k=(V_k, E_k) \), we define the longest path of task \( \tau_k \) as a sequence of nodes \( \lambda=(v_{k,a}, v_{k,a+1}, \ldots, v_{k,b}) \) from start node \( v_{k,a} \) to node \( v_{k,b} \). \( v_{k,b} \) represents task \( \tau_k \). The longest path of \( \tau_k \), denoted by \( lp(\lambda) \), is the sum of the WCETs of all its nodes, that is,

\[
lp(\lambda) = \sum_{j=a}^{b-1} ET_{i,j}
\]

A necessary condition for the schedulability of task \( \tau_k \) is thus \( lp(\lambda) \leq D_k \).

5. 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. Task schedulability is defined as the number of completed tasks divided by the total number of tasks.

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 DAG task set generation algorithm that is used in Ref. [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 3 shows the schedulability rate of tasks with different number of cores under PRTTS and GEDF scheduling strategies when memory bank \( m=2 \). It can be seen that the schedulability of tasks in PRTTS scheduling strategy is higher than that 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 task.

Figure 4 shows the schedulability rate of task sets with different number of cores under PRTTS and GEDF scheduling strategies when memory bank \( m=2 \) and the task number \( u=2 \sim 8 \). 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 3. Schedulability rate of tasks with different number of cores ($m=2$).

Figure 4. Schedulability rate of task sets with different number of cores ($m=2$, $u=2\sim8$).

6. Conclusion
In this paper, we propose PRTTS scheduling strategy for parallel real-time tasks. In PRTTS scheduling strategy priorities of tasks that access memory are promoted over priorities of tasks not accessing memory. Tasks accessing the data in the cache is dynamically raised to highest priority. The results of simulation experiment show that proposed new scheduling strategy offers better performance in terms of schedulability rate of tasks.

References
[1] Lei X D, Zhao Y L and Yuan X L 2009 Concurrency control in mobile distributed real-time database systems Journal of Parallel and Distributed Computing 69 (10) 866-876.
[2] Alessandra M, Marko B, Vincenzo B, Alberto M S and Giorgio B 2017 Schedulability analysis of conditional parallel task graphs in multicore systems IEEE Transactions on Computers 66 (2) 339-353.
[3] Heechul Y, Gang Y, Rodolfo P, Marco C and Lui S 2016 Memory bandwidth management for efficient performance isolation in multi-core platforms IEEE Transactions on Computers 65 (2) 562-576.
[4] Lee J and Shin K G 2012 Controlling preemption for better schedulability in multi-core systems Proc. IEEE 33rd Real-Time Syst. Symp. pp 29-38.
[5] Rosen J, Andrei A, Eles P and Peng Z 2007 Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip Proc. 28th IEEE Int. Real-Time Syst. Symp. pp 49-60.

[6] Gang Y, Rodolfo P, Stanley B, Heechul Y and Marco C 2016 Global real-time memory-centric scheduling for multicore systems IEEE Transactions on Computers 65 (9) 2739-2751.

[7] Gang Y, Pellizzoni R and Bak S 2012 Memory centric scheduling for multicore hard real-time systems Real-Time System 48 (16) 681-715.

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

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

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

[11] Hoon S C, Lee J, Phan K M, Arvind E and Insik Shin 2017 Global EDF schedulability analysis for parallel tasks on multi-core platforms IEEE Transactions on Parallel and Distributed Systems 28 (5) pp 1331-1344.

[12] Abusayeed S, David F, Li J, Kunal A, Lu C and Christopher D G 2014 Parallel real-time scheduling of DAGs IEEE Transactions on Parallel and Distributed Systems 25 (12) 3242-3252.