ATLANTIS PRESS

Advances in Computer Science Research, (ACSR), volume 52

2016 International Conference on Computer Engineering and Information Systems (CEIS-16)

Task-Aware Priority Scheduling for Multicore Processors

Qiu-Wei Shi
School of Software, Shanghai Jiao Tong University
Shanghai, China
E-mail: LouisXivR@outlook.com

***Abstract*-Task scheduling is a key issue in multicore processors, especially for the multicore processor in parallel framework. However, few state-of-art task scheduling strategies solve these important issues simultaneously: assignment of task to CPU core, task types, task order and resource conflict of task. To the best of our knowledge, the schedule problem is more complicated in the node of parallel framework. To settle this problem, we propose a method of task-aware priority scheduling for multicore processors. First, we introduce our scheduling framework, including four modules: Kernel Scheduler, Resource Management, Synchronization, Node Coordinator. Then follow two key algorithms: Task-to-core assignment algorithm and Resource distribution algorithm. Finally, we show that our method obtains good performance in contrast with Work-stealing algorithm.**

Key words: Task Scheduling; Multicore Processors

I. INTRODUCTION

Task scheduling is a key issue in multicore processors, especially for the multicore processor in parallel framework. Massive studies put forward with novel task scheduling strategies in terms of the task execution order, the task energy consumption, whether the memory is shared or not, and so on.

As we concern, scheduling strategies make a considerable difference between standalone multicore processors and one node in parallel framework, especially in the following aspects.

A. *Task Type.*

Task can be divided into two categories: interactive task and non-interactive task. Interactive task communicates with other tasks in terms of orders and task data, while non-interactive task can run individually. When tasks come from the scheduler in parallel framework, they are always of different types and of different applications, as Fig. 1 shows. By comparison, most of tasks on standalone multicore processors are of the same type, interactive or non-interactive, and of the same application.

B. *Task Order.*

It is more unpredictable about the task order on the node in parallel framework. The task order may have four cases: interactive task coming after interactive task, non-interactive task coming after interactive task, interactive task coming after non-interactive task and non-interactive task coming after non-interactive task. Besides, tasks of different application always come to the node in parallel

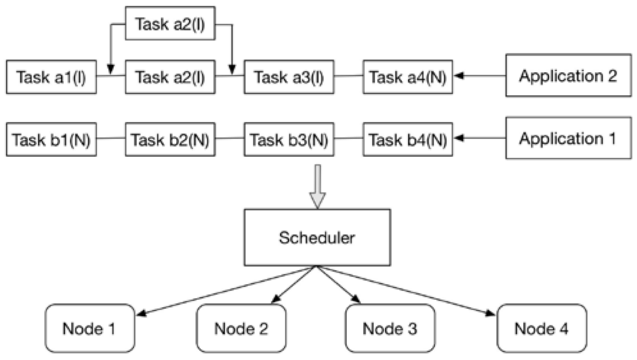

Figure 1. Task scheduling in parallel framework.

framework randomly, while tasks of the same application come together on standalone multicore processors.

C. *Resource Race.*

The resource race between tasks is more complex on the node in parallel framework. Since tasks are always of different types and of different applications, resource races exist not only between tasks in the same node, but also among tasks of the same application on different nodes. However, standalone multicore processors only need to consider the first situation.

D. *Synchronization.*

Similar to resource race, synchronization on one node in parallel framework, can be divided into two part: synchronization between tasks in the same node, synchronization between tasks of the same application on different nodes. The second part is tougher to formulate a proper strategy. By contrast, standalone multicore processors only need to consider the first situation.

To solve these issues, we propose a method of task-aware priority scheduling for multicore processors that compute thread priority dynamically and achieve a well balance between thread priority and performance. The key contribution of our work is as follows:

- We present a task-aware priority scheduling strategy that solve these issues simultaneously: the assignment of task to CPU core, task types, task order and resource conflict of task. To the best of our knowledge, the schedule problem is more complicated in every node of parallel framework and no previous work to solve these issues simultaneously.

- We formulate a scheduling framework and two key algorithms: Task-to-Core Assignment Algorithm and Resource Distribution Algorithm. We evaluate our approach on a multi-core x86 machine, in contrast with Work-stealing algorithm.

The reminder of this paper is organized as follows. In the next section, we describe some task information. In Section 3, we present the framework of task-aware priority scheduling and the two key algorithms: Task-to-Core Assignment Algorithm and Resource Distribution Algorithm. In Section 4, we evaluate our method, and make a brief comparison with the classic scheduling algorithm, Work-stealing algorithm. Section 5 present several relevant researches. And Section 6 contains our concluding remarks.

II. TASK MODEL

An application is divided into an amount of tasks as a task set, which is represented as TaskS. A task set i with all tasks is represented as TaskSi = {Task1, Task2, ...}. Every node of parallel framework contains several on-going applications and only part of tasks for each of them. These tasks are of different types: interactive and non-interactive, represented as TaskI and TaskN. A waiting queue in one node of parallel framework, contains tasks of different types and of different applications, which is marked as WaitQ = {TaskI1, TaskI2, TaskN1, TaskN2, TaskN3, ...}. The relationship between task and applications is recorded in a task table on every node.

III. SCHEDULING STRATEGY

A. *Scheduling framework*

We propose a task-aware priority scheduling strategy which performs well on dealing with t asks of different types and of different applications in one node of parallel framework. To formulate the scheduling strategy, four modules are designed as Fig. 2 shows.

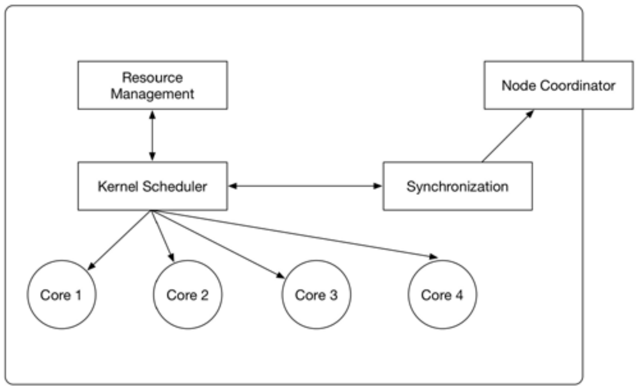

Figure 2. A multicore node in parallel framework.

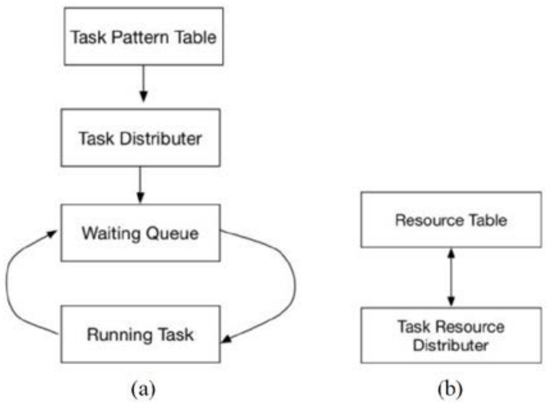

Figure 3. Framework modules: (a) Kernel Scheduler, (b) Resource Management.

1) Kernel Scheduler: Fig. 3(a) shows that kernel scheduler keeps four main part: Task Pattern Table, Task Distributer, Waiting Queue and Running Task.Task Pattern table provides a method to identify the task type. Task Distributer computes the task priority of every task and insert tasks into waiting queue. Waiting queue is a priority-constrained task queue. Running Task refers to the on-going tasks.

2) Resource Management: This module keeps a Resource Table and distributes resource among tasks. As Fig. 3(b) shows, Resource Management contains two parts, among which Resource Table records the rest amount of resources and the amount of resource distributed to every running task, and Task Resource Distributer computes the amount of resource for every task in the waiting queue.

3) Synchronization: Synchronization records every task information in the node, such as task type, task status, the application to which the task belongs, task generation time and so on. It also records failure reason for the tasks and report tasks to scheduler in parallel framework for the failed task due to lack of local resource, by node coordinator.

4) Node Coordinator: Node Coordinator receives failure report from Synchronization and communicates with scheduler in parallel framework.

B. *Scheduling Alogorithm*

We put forward with two key algorithms in scheduling framework.

1) Task-to-core assignment algorithm: When there exist idle cores, task is assigned to core according to the rule, first in first run, just as Fig. 4 shows. When there are no idle cores. It is more complex. There exist several steps for every $Task_i$, as Fig. 5 shows.

a) Identify task type, marked as $Task_i$.TaskT: As the above mentioned, we divide tasks into two classes: interactive and non-interactive. Task pattern table identifies task type for every task.

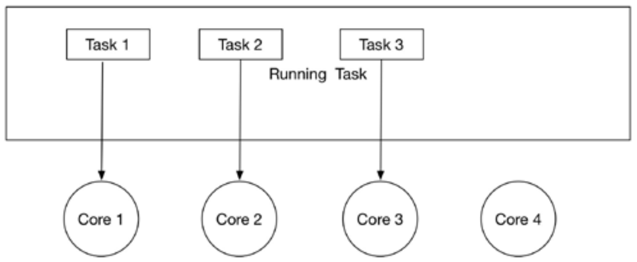

Figure 4. Task-to-core assignment (FIFR)

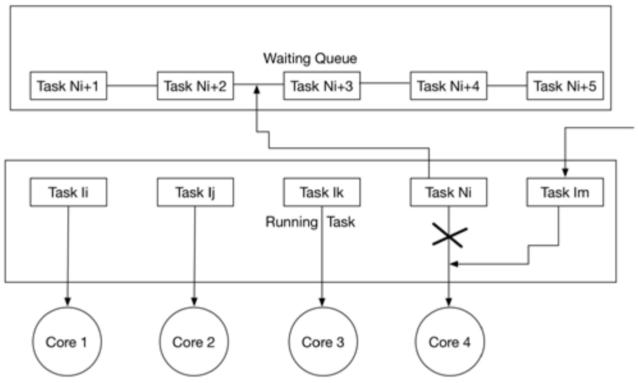

Figure 5. Task-to-core assignment (PR)

b) Calculate task priority, marked as $Task_i$.TaskP: Task priority is influenced by task type, the task left time and task running times, which is given by

$$Taski.TaskP = \alpha * Taski.TaskT + \beta * Taski.TaskLeftT + \gamma * Taski.RunningT \qquad (1)$$

The task left time for every task is influenced by the belonged application id, task generation time on the node. We set three parameters alpha, beta and gamma in line with our original intention that interactive task has a higher priority than non-interactive task and the task which fails or is terminated by other task before has a higher priority than the task in its first time.

c) Compare the task priority of $Task_i$ with that of current running task: If higher, current running task is terminated and inserted into waiting queue with updated priority and the waiting queue index, while $Task_i$ is assigned to the core. Otherwise, calculate waiting queue index for $Task_i$, denoted by $Task_i$.WQIndex and insert it into waiting queue.

2) Resource distribution algorithm: Massive researches consider task isolation and and consumption prediction of task required resource. However, few reaches a good balance between program performance with prediction accuracy. Resource distribution algorithm in this paper, put forward with a conception, resource maximum, denoted by *$Task_i$.resourceMax*. For every task which is ready to run on core, calculate resource maximum and set the upper resource bound during resource distributing. If succeed,

Algorithm 1 Task-to-core assignment algorithm

```
procedure TASK-TO-CORE($Task_i$ comes)
    if there exists idle $core_j$ then        ▷ First in first run, FIFR
        $core_j \leftarrow Task_i$
    end if
    if there is no idle cores then ▷ Pre-emptible run, PR
        identify $Task_i.TaskT$
        calculate $Task_i.TaskP$
        if $Task_i.TaskP > RunningTask.TaskP$ then
            re-calculate $RunningTask.TaskP$
            calculate $RunningTask.WQIndex$
            terminate RunningTask and put it back to Waiting Queue
            $core \leftarrow Task_i$
        end if
        calculate $Task_i.WQIndex$
        insert $Task_i$ into Waiting Queue
    end if
end procedure
```

$Task_i$ run successfully on the core. Otherwise, re-calculate *$Task_i$.TaskP* and re-insert it into waiting queue. *$Task_i$.resourceMax* is given by

$$Task_i.resourceMax = resourceRest / (idleCoreN + k * Task_i.runningT + b) \qquad (2)$$

The rest amount of resource is denoted by *resourceRest*. The number of idle cores is denoted by *idleCoreN*.

Algorithm 2 Resource distribution algorithm

```
procedure RESOURCE DISTRIBUTION($core_j \leftarrow Task_i$)
    calculate $Task_i.resourceMax$
    if $Task_i.resourceNeed <= Task_i.resourceMax$ then
        $Task_i$ run successfully on $core_i$
    end if
    if $Task_i.resourceNeed > Task_i.resourceMax$ then
        re-calculate $RunningTask.TaskP$
        calculate $RunningTask.WQIndex$
        put $Task_i$ back to Waiting Queue
    end if
end procedure
```

IV. EVALUATION

In this section, we evaluate our scheduling strategy for Parallel Shortest Path Algorithm and the following benchmarks: perlbench, bzip, gcc, mcf, gobmk, hmmer, sjeng, libquentum. First, we analyses task priority with different parameter values, alpha: beta: gamma. Then, we compare the performance of task-aware priority scheduling with Work-stealing algorithm.

A. Experiment 1

This experiment mainly analyses the influence of the different parameter values, alpha: beta: gamma. As we mentioned in Section 3, the three parameters alpha, beta, gamma, have a significant impact on the task priority, which directly influences scheduling result for every task, leading

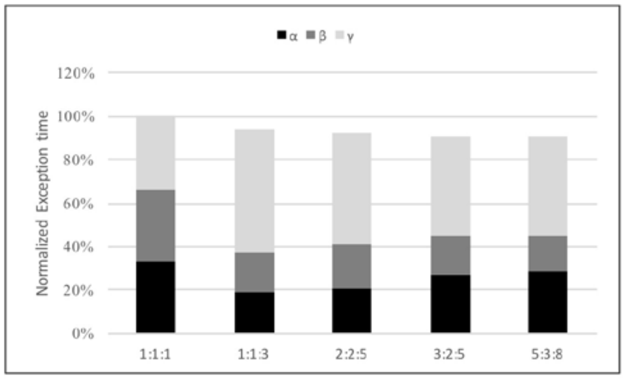

Figure 6. Different parameter values *α*: *β*: *γ*.

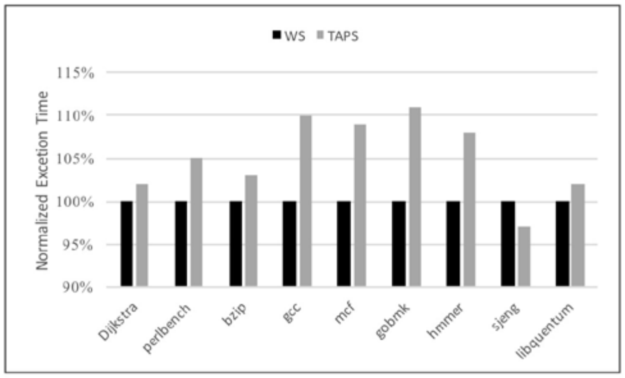

Figure 7. Work-stealing (WS) vs Task-Aware Priority Scheduling(TAPS).

to different task execution order. In this experiment, we have tested more than dozens of different parameter values and picked out five groups of values, which is shown in Fig. 6. Compared to 1:1:1, we conclude that 5:3:8 obtains a biggest decline by 9% in execution time for our test application, which means that the running times of task has the greatest influence on task priority and the task left time ranks last.

B. Experiment 2

This experiment makes a comparison between Work-stealing algorithm and our approach. In this experiment, the results of our method is based on the conclusion of experiment 1 that task priority equation is fixed with α: β: γ =5:3:8. As Fig. 7 shows, our approach has more execution time by 5% on average, in contrast with Work-stealing algorithm. The results prove that our approach obtains a good balance between performance and requirements.

V. RELATED WORKS

With the fast development of computer hardware and software technology, task scheduling on multicore processors becomes a hot topic in parallel computing. Among massive relevant researches, scheduling strategy focuses on two points: historical data learning and dynamic modeling.

A. Historical Data Learning

Several studies address proper scheduling strategy on multicore processors based on historical data learning, such as task execution order. Deterministic multithreading scheduling problem is a class of multitasking scheduling problem on multicore processors. Deterministic multithreading expects to obtain the same output for the same input. Kendo [1], a multithreaded scheduling system for parallel application, records and reuse the deterministic logical time for every thread before, by distributing resources and organizing thread execution order in accordance with it. PEREGRINE [2] ensures the determinacy in multithreaded scheduling procedure through saving the execution path of the whole program and reusing the scheduling strategy before. Lu [3], Bond [4], Mushtaq [5] also focus on similar studies.

Radojkovicp [6] put forward with a kind of system-level scheduling strategy, BlackBox, which maintains a Base time table and a Slowdown table based on historical scheduling strategy. The Base time table records the performance value for every scheduling decision, while Slowdown table shows the performance drop-out value due to thread race. when BlackBox makes scheduling decision, it considers two tables to make performance prediction and picks out the best solution. In new research [7], Radojkovicp presents the best scheduling scheme by combining Extreme Value theory and performance evaluation of historical scheduling result.

However, these researches are based on that the same applictaion runs multiple times or that application type is within several types. As for the node in parallel framwork, diverse application types and less repetition of applications make it less meaningful to study massive historical data, maintain a large amount of intermediate result and reuse seldom part of historical result.

B. Dynamic Modeling

By comparison, more researches focus on dynamic modeling when making concrete scheduling decision. Multithreaded scheduling of real-time systems is a key class of multitasking scheduling problem on multicore processors. Sarker [8] divides task scheduling into two stages: task selection and task distribution. Every time, it picks out the tasks which have a higher probability of resource allocation and assigns them to cores as little as possible with coloring principle. This method makes the best use of core resources and deploys more subsystems on systems with a fixed number of cores. With consideration for task deadline, Lin [9] builds dynamic energy model and realize the minimum energy consumption of DVFS multicore processors. In addition, Rayo [10] also realizes DVFS with heuristic method.

Kim [11] solves Integer Linear programming dynamically to keep the worst execution time within an acceptable scale. Petrucci [12] also optimizes Integer Linear programming. Alhammad [13] presents Fork-join parallel model, making the most of divider and conquer. Without taking into account the constraint relations between threads, they dynamically adjust thread execution order to shorten the running time of the entire application.

In this paper, our research fully utilizes the idea of dynamic modeling, develop scheduling strategy based on calculating task priority dynamically.

VI. CONCLUSION

In this paper, we propose a method of task-aware priority scheduling for multicore processors that solves these issues simultaneously: the assignment of task to CPU core, the thread types, the order and resource conflict of thread. Then we make a brief comparison with the classic scheduling algorithm, Work-stealing algorithm, which proves that our research obtains a good balance between performance and requirements.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This research was sponsored in part by 863 Program in China (Grant No. 2014AA01A301), the National Nature Science Foundation of China (Grant No. 61572312 and 61572313), and Science and Technology Commission of Shanghai Municipality's Innovation Action Plan (No. 15DZ1100305).

REFERENCES

[1] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009, pages 97–108, 2009.

[2] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011, pages 337–351, 2011.

[3] K. Lu, X. Zhou, X. Wang, T. Bergan, and C. Chen. An efficient and flexible deterministic framework for multithreaded programs. J. Comput. Sci. Technol., 30(1):42–56, 2015.

[4] M. D. Bond, M. Kulkarni, M. Cao, M. F. Salmi, and J. Huang. Efficient deterministic replay of multithreaded executions in a managed language virtual machine. In Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015, Melbourne, FL, USA, September 8-11, 2015, pages 90–101, 2015.

[5] H. Mushtaq, Z. Al-Ars, and K. Bertels. Efficent and highly portable deterministic multithreading (detlock). Computing, 96(12):1131–1147, 2014.

[6] P. Radojkovic, V. Cakarevic, J. Verdu ́, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread assignment of multithreaded network applications in multicore/multithreaded processors. IEEE Trans. Parallel Distrib. Syst., 24(12):2513–2525, 2013.

[7] P. Radojkovic, P. M. Carpenter, M. Moreto ́, V. Cakarevic, J. Verdu ́, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread assignment in multicore/multithreaded processors: A statistical approach. IEEE Trans. Computers, 65(1):256–269, 2016.

[8] A. Sarkar, F. Mueller, and H. Ramaprasad. Static task partitioning for locked caches in multicore real-time systems. ACM Trans. Embedded Comput. Syst., 14(1):4:1–4:30, 2015.

[9] C. Lin, C. Chang, Y. Syu, J. Wu, P. Liu, P. Cheng, and W. Hsu. An energy-efficient task scheduler for multi-core platforms with per-core DVFS based on task characteristics. In 43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014, pages 381–390, 2014.

[10] D. B. Rayo, J. S. Borra ́s, H. H. Mohamed, S. Petit, and J. Duato. Balancing task resource requirements in embedded multithreaded multicore processors to reduce power consumption. In Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010, pages 200–204, 2010.

[11] Y. Kim, D. Broman, J. Cai, and A. Shrivastava. Wcet-aware dynamic code management on scratchpads for software-managed multicores. In 20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014, pages 179–188, 2014.

[12] V. Petrucci, O. Loques, D. Moss ́e, R. G. Melhem, N. A. Gazala, and S. Gobriel. Energy-efficient thread assignment optimization for heterogeneous multicore systems. ACM Trans. Embedded Comput. Syst., 14(1):15:1–15:26, 2015.

[13] A. Alhammad and R. Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014, pages 1–6, 2014.