A Hybrid Real-Time Scheduling Approach on Multi-Core Architectures

Pengliu Tan, Jian Shu and Zhenhua Wu
School of Software, Nanchang Hangkong University, Nanchang, China
Email: Pengliu.Tan@gmail.com

***Abstract*—This paper proposes a hybrid scheduling approach for real-time system on homogeneous multi-core architecture. To make the best of the available parallelism in these systems, first an application is partitioned into some parallel tasks as much as possible. Then the parallel tasks are dispatched to different cores, so as to execute in parallel. In each core, real-time tasks can run concurrently with non-real-time tasks. The hybrid scheduling approach uses a two-level scheduling scheme. At the top level, a sporadic server is assigned to each scheduling policy. Each sporadic server is used to schedule the dispatched tasks according to its scheduling policy. At the bottom level, a rate-monotonic OS scheduler is adopted to maintain and schedule the top level sporadic servers. The schedulability test is also considered in this paper. The experimental results show that the hybrid scheme is an efficient scheduling scheme.**

***Index Terms*—scheduling, real-time, multi-core**

I. INTRODUCTION

Current chip fabrication technologies allow placing several million transistors in a chip, enabling more complex designs each time. However, there are several issues that discourage the design of more complex uniprocessors: the increase in heat generation, the diminishing instruction-level parallelism gains, almost unchanged memory latency, the inherent complexity of designing a single core with a large number of transistors and the economical costs derived of this design. For these reasons, the current trend on chip manufacturing is to place multiple processor cores (multi-core) on a chip[1].

A multi-core processor (or chip-level multiprocessor, CMP) combines two or more independent cores (normally a CPU) into a single package composed of a single integrated circuit (IC), called a die, or more dies packaged together. A dual-core processor contains two cores, and a quad-core processor contains four cores. Cores in a multi-core device may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD dual-core processors). The processors also share the same interconnect to the rest of the system. Each "core" independently implements optimizations such as superscalar execution, pipelining, and multithreading. A system with *n* cores is effective when it is presented with *n* or more threads concurrently.

The amount of performance gained by the use of a multi-core processor depends on the problem being solved and the algorithms used, as well as their implementation in software (Amdahl's law [2]). For so-called "embarrassingly parallel" problems, a dual-core processor with two cores at 2GHz may perform very nearly as quickly as a single core of 4GHz. Other problems, though, may not yield so much speedup. This all assumes, however, that the software has been designed to take advantage of available parallelism. If it hasn't, there will not be any speedup at all. However, the processor will multitask better since it can run two programs at once, one on each core.

In addition to operating system (OS) support, adjustments to existing software are required to maximize utilization of the computing resources provided by multi-core processors. Thereinto, the task scheduling is the most important in the multi-core systems, especially to the real-time systems with multi-core. Also, the ability of multi-core processors to increase application performance depends on the use of multiple threads within applications.

In recent years, some research has been done in the field about the real-time scheduling on multi-core architecture. In [3], James H Anderson, et al. proposed a cache-aware Pfair-based scheduling scheme for real-time tasks on multi-core platforms, but they had only considered static and independent tasks. The authors of [4] discussed an approach for supporting soft real-time periodic tasks in Linux on performance asymmetric multi-core platforms, or AMPs. In [5], a scheduling method was proposed for real-time systems implemented on multi-core platforms that encourage individual threads of multithreaded real-time tasks to be scheduled together. In [6], the authors considered the problem of scheduling soft real-time workloads on such a heterogeneous multi-core platform with faster and slower cores. The authors of [7] proposed a technique, based on dynamic on-line reconfiguration of a four-processor multi-core hardware platform, to achieve a tradeoff between performance (through parallelism) and fault-tolerance (through replication), for real-time systems. In [8], a hybrid approach was proposed for scheduling real-time tasks on large-scale multi-core platforms with hierarchical shared caches. In this approach, a multi-core platform was partitioned into clusters. Tasks were statically assigned to these clusters, and scheduled within each cluster using the

Manuscript received August 10, 2009; revised September 10, 2009; accepted September 19, 2009.
This paper is supported by National Science Foundation of China under grant 60773055.

© 2010 ACADEMY PUBLISHER
doi:10.4304/jsw.5.9.958-965

preemptive global EDF scheduling algorithm. The paper [9] tackled the problem of reducing power consumption in a periodic real-time system using DVS on a multi-core processor. In paper [10], with task migration, the authors first presented a novel-model called T-L_{er} plane to describe the behavior of tasks and processors, and two optimal on-line algorithms based on T-L_{er} plane to schedule real-time tasks with dynamic-priority assignment on uniform multiprocessors. To make it practical and to reduce context switches, they also presented a polynomial-time algorithm to bound the times of rescheduling in a T-L_{er} plane. In [11], the scalability of the scheduling algorithms used to support real-time workloads on multi-core platforms was considered and an empirical evaluation of several global and partitioned scheduling algorithms was presented. The authors of [12] proposed a novel soft power-aware real-time scheduler for a state-of-the-art multi-core multithreaded processor, which implemented dynamic voltage scaling techniques. The proposed scheduler reduced the energy consumption while satisfying the constraints of soft real-time applications. In [13], various heuristics were explored that attempt to improve cache performance when scheduling real-time workloads on multi-core platforms. Such heuristics were applicable when multiple multithreaded applications exist with large working sets. In addition, a case study was presented.

But the research in [3-13] had not considered the co-scheduling of real-time application with non-real-time applications, and also not separated the scheduling mechanism from the scheduling policy.

The author of [14] presented an approach for supporting soft real-time periodic tasks and non-real-time tasks. It used a two-level hierarchical scheduling method that decreases average deadline miss ratio. It could not only maintain the response time of general tasks, but also support the real-time requirements for other tasks. In addition, the Pfair class of algorithms that allow full migration and fully dynamic priorities had been shown to be theoretically optimal. But, it incurred significant run-time overhead due to their quantum-based scheduling approach. Moreover, hard real-time tasks were not considered in this paper.

To resolve these problems efficiently, this paper proposes a two-level scheduling scheme, which uses the idea of sporadic servers [15] and extends the useful concept of open systems proposed in [16-18], where the schedulability of each real-time application can be validated independently of other applications in the system. But the method proposed in [16-18] could not support the parallelism between the tasks within an application. The scheduling scheme proposed in this paper can exploit the parallelism between the tasks within an application efficiently. In addition, the homogeneous multi-core architectures are only considered in this paper.

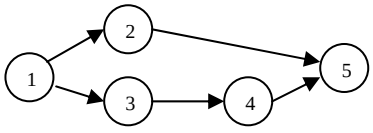
The rest of the paper is structured as follows: the task partitioning is discussed in Section II. The architecture of the scheduling scheme is described in Section III. The budget replenishment mechanism is introduced in Section IV. Section V develops the schedulability tests for real-time tasks. Section VI presents performance evaluation. Finally, in Section VII, some concluding remarks are made.

II. TASK PARTITIONING

A. Task Model

Each application consists of several tasks with or without precedence constraints. Each real-time application $A(r, d)$ is characterized by a release time r and a relative deadline d. They are known at release time. Each real-time task $\tau_i(r_i, c_i, e_i, l_i, d_i)$ is characterized by a release time r_i, a computation time c_i, the earliest start time e_i, the latest start time l_i and a relative deadline d_i. The release time and computation time are known at the release time of its application. The earliest start time, the latest start time and the relative deadline are got by computing according to the precedence relationship between tasks.

To scheduling the tasks with precedence constraints, we partition them into the independent tasks first. The precedence relationship is denoted by the task graph which is directed and no-loop. We assume that the tasks between two different applications are independent and the communication overhead between tasks due to their precedence relationship is considered in task's computation time. For example, an application $A(r=0, d=10)$ consists of five tasks, and their precedence relationship can be shown as Figure 1.



τ_1: $r_1=0$, $c_1=1$; τ_2: $r_2=0$, $c_2=2$; τ_3: $r_3=0$, $c_3=1$;
τ_4: $r_4=0$, $c_4=3$; τ_5: $r_5=0$, $c_5=1$;

Figure 1. An example: Task Graph

B. Partitioning Algorithm

We will partition the tasks with precedence constraints into the independent tasks according to the algorithm shown in Figure 2.

Step 1. Topology sort and compute the earliest start time.

- Firstly, initialize in-degree array according to task graph and initialize the earliest start time of each task to its release time.
- Secondly, push the node with zero in-degree into original stack.
- Thirdly, if the original stack is not empty, pop the top node (denoted by *CurNode*) in the original stack and push it into the topology-sort stack and search the first adjacent node *nextadj* of *CurNode*. If the original stack is empty, go to step 2.
- Fourthly, if *nextadj* exists, the in-degree of *nextadj* is decreased by 1. If the in-degree of *nextadj* is zero, push *nextadj* into the original

© 2010 ACADEMY PUBLISHER

stack. If the sum $\sum$ of the earliest start time and the computation time of the task corresponding to *CurNode* is larger than the earliest start time of the task corresponding to *nextadj*, set the earliest start time of the task corresponding to *nextadj* to $\sum$. And then search the next adjacent node *nextadj*. Repeat the step until any adjacent node cannot be found.

- Fifthly, if *nextadj* does not exist, go to the third step.

Step 2. Compute the latest start time.

- Firstly, initially, set the latest start time of each task to (*D-c*), wherein, *D* denotes the absolute deadline of the application including the task and *c* denotes the computation time of the task.
- Secondly, if the topology-sort stack is not empty, pop the top node (denoted by *CurNode*) in the topology-sort stack and search the first adjacent node *nextadj* of *CurNode*. If the topology-sort stack is empty, go to step 3.
- Thirdly, if *nextadj* exists, continue, or go to the second step. Assume the computation time of the task corresponding to *CurNode* subtracted from the latest start time of the task corresponding to the *nextadj* gives *l*. If *l* is less than the latest start time of the task corresponding to to *CurNode*, set the latest start time of the task corresponding to *CurNode* to *l*. And then search the next adjacent node *nextadj*. Repeat the step until any adjacent node cannot be found.

Step 3. Compute the relative deadline.

A task $\tau_i(r_i, c_i, e_i, l_i, d_i)$, set d_i to $l_i - e_i + c_i$. Wherein, r_i, c_i, e_i, l_i and d_i denote the release time, the computation time, the earliest start time, the latest start time and the relative deadline of the task τ_i respectively.

Figure 2. Partitioning Algorithm

To minimize the communication overhead, the tasks with serial relationship are dispatched to the same core as much as possible. In addition, to effectively exploit the available parallelism and balance the workload in these systems, we use load balancing allocation algorithm, and the tasks which can run in parallel are assigned to different cores as much as possible. For example shown in Figure.1, the task τ_1, τ_2 and τ_5 can be dispatched to a core and the others to another core.

C. Load Balancing Allocation Algorithm

The load balancing allocation algorithm consists of two sub-algorithms: Most-Demand-First Algorithm (MDFA) and Idlest-Fit Algorithm (IFA). The main idea of MDFA is that the task with the most CPU utilization demand is allocated first, and the tasks in an application are sorted in the decreasing order of their reserved CPU utilizations. The idea of IFA is that the processing cores are sorted in the decreasing order of their available CPU capacity, and the idlest processing core is always searched first when an allocation decision is made. MDFA resolves the problem which task is first allocated, however, IFA resolves the problem which processing core a task is first allocated to. MDFA and IFA make up of load balancing allocation algorithm.

Suppose a just submitted application A_x has n tasks (t_1, t_2, ……, t_n). The reserved CPU utilization of t_j is u_j. There are m processing cores (P_1, P_2, ……, P_m) in the multi-core system. The remainder CPU utilization capacity of P_j is C_j (C_j >0). In all, there are the product of n factorial and m factorial allocation ways and maybe the product of n factorial and m factorial allocation results. If the n tasks in A_x can be allocated successfully, there is a necessary condition:

The maximum of u_1, u_2, ……, u_n must be less than or equal to the maximum of C_1, C_2, ……, C_m.

MDFA and IFA can judge at the fastest rate whether the necessary condition is met or not. So it is an efficient allocation method. Further, it always allocates the task with the most CPU utilization demand to the processing core with the most available CPU utilization capacity first, so it is also a good load balancing allocation method.

The steps of MDFA and IFA are as follows:

Step 1 Sort the tasks in the just submitted application A_x in the decreasing order of their reserved CPU utilizations.

Step 2 Sort the processing cores in the decreasing order of their available CPU utilization capacities.

Step 3 Compare the reserved CPU utilization u of the first task with the available CPU utilization capacity C of the first processing core. If $u \leq C$, do schedulability test, or reject the application A_x and end the allocation for the application A_x. If the task is schedulable, pre-allocate the task to the processing core and go to step 2, or reject the application A_x and end the allocation for the application A_x.

If any task in A_x is pre-allocated to one processing core successfully, allocate it to the processing core really and end the allocation procedure.

III. SCHEDULING ARCHITECTURE

The workload in the real-time system consists of real-time (hard and soft) and non-real-time tasks. In each processing core, there are several sporadic servers with a CPU budget c and a period p at the top level. Each sporadic server is associated with a ready queue contains ready tasks to run on the server. Each server has a scheduler associated with it. The server scheduler uses a scheduling algorithm, such as EDF or RM or time sharing algorithm, to schedule tasks and order tasks in the ready queue of sporadic server. All the real-time tasks with the earliest deadline first (EDF) algorithm are executed on the sporadic server S_{EDF}, all the real-time tasks with the rate monotonic (RM) algorithm are executed on the sporadic server S_{RM}, and so on. In addition, all the non-real-time tasks, which adopt the time sharing scheduling policy, are executed on the sporadic server S_{TS}.

At the bottom level, there is a fixed-priority driven scheduler (RM scheduler) called the operating system

© 2010 ACADEMY PUBLISHER

scheduler, which is adopted to maintain all the top level sporadic servers. The OS scheduler replenishes the server budget for every server according to the definitions of sporadic server [15]. A server is ready if its ready queue is not empty. The OS scheduler schedules all the ready servers according to the rate monotonic (RM) scheduling algorithm. Tasks of the scheduled ready server can execute until they run out of the budget of the server, or a higher-priority ready server arrives. Tasks of the scheduled ready server execute in the order defined by the chosen scheduling algorithm of the server.

Figure 3 (a) shows the task partitioning and allocation on multi-core architecture. Figure 3 (b) shows the scheduling architecture in processing core.

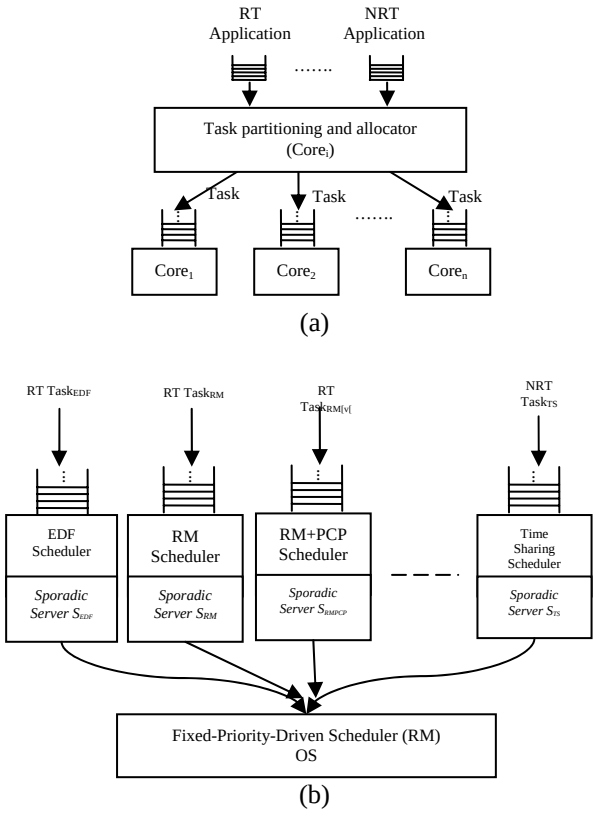


Figure 3. a) Task partitioning and allocation on multi-core architecture. b) Scheduling architecture in processing core.

The operations of the OS scheduler are shown as Figure 4.

Initiation:

- Create a sporadic server S_δ for each scheduling policy in the system, such as a S_{EDF} for the real-time tasks with EDF algorithm, a sporadic server S_{RM} for the real-time tasks with RM algorithm, and a sporadic server S_{TS} for the non-real-time tasks.

Maintenance of server S_δ:

- Replenish the server budget according to the definitions of sporadic server [15].
- The scheduled task of each server S_δ executes under the CPU budget of S_δ.

Interaction with server scheduler of each server S_δ:

- The scheduler of each server S_δ schedules tasks according to the chosen algorithm $_\delta$.

Scheduling of all servers:

- The OS scheduler schedules the ready server with the highest priority in the system.

Figure 4. Operations of the OS scheduler in PN

IV. BUDGET REPLENISHMENT MECHANISM

The budget replenishment mechanism [17] of a sporadic server is summarized as follows:

Each sporadic server is associated with a CPU budget c and a period p. Suppose that the system is scheduled by a fixed-priority scheduler. Let P_s denote the priority of the task which is executing. A priority level P_i is active if $P_s \geq P_i$. A priority level is idle if it is not active. Let RT_i denote the replenishment time for a sporadic server executing at priority level P_i. The replenishment time RT_i of server S_i (with priority level P_i and period p_i) is set as follows:

- If S_i has a non-zero remaining CPU budget, and P_i becomes active at time t, then $RT_i = t + p_i$.
- If the CPU budget of S_i is exhausted, and the CPU budget of S_i is replenished at time t, then $RT_i = t + p_i$.

The replenishment amount is determined when P_i becomes idle, or when the remaining CPU budget of S_i becomes zero. The replenishment amount is equal to the amount of server execution time consumed since the last time at which the status of P_i changes from idle to active. As shown in [15], a periodic task set that is schedulable with a periodic task τ_i is also schedulable if τ_i is replaced with a sporadic server with the same period and CPU budget. The schedulability analysis of sporadic servers is equivalent to that of periodic tasks. We refer interested readers to [15] for details.

V. SCHEDULABILITY CONDITION

The system always admits non-real-time applications, but it admits a real-time application into the system only when all the real-time tasks in the application meet the schedulability condition. If all the real-time tasks in A_k are schedulable, A_k is schedulable, or A_k is unschedulable. So the schedulability of real-time application becomes the schedulability of its tasks.

First, we give several definitions.

Definition 1 *Sporadic server size* is the ratio of server budget c to server period p.

Definition 2 *Processor utilization factor* is the ratio of the execution time of a periodic task to its period or the ratio of the execution time of an aperiodic task to its relative deadline.

Definition 3 *The worst-case blocking time of task τ_i* is the worst-case execution time of the critical section of other tasks sharing resource with task τ_i.

© 2010 ACADEMY PUBLISHER

A. Schedulability Condition of Sporadic Servers According to RM Algorithm

We say that a sporadic server is schedulable if the server budget is always exhausted at or before its deadline after every time its budget and deadline are set. To state this fact in another way, we can view each sporadic server as a periodic task with execution time equal to the server budget c and period equal to the server period p which is released each time the server budget is replenished.

Theorem 1: *If the total size U of all the sporadic servers in processing core is less than or equal to $k(2^{1/k}-1)$, i.e.,*

$$U = \sum_{i=1}^{k} u_i \le k(2^{1/k} - 1)$$

the servers according to RM algorithm are schedulable. Therein, k is the number of sporadic servers in the processing core and u_i is the server size of sporadic server i.

Proof: For a given set of k independent and preemptive tasks with fixed priority order, they are schedulable if and only if the total processor utilization factor Σ of the k tasks is less than or equal to $k(2^{1/k}-1)$ according to the theorem 4 in [19]. In our system, each sporadic server is looked as a special periodic task with execution time c and period p. In addition, the tasks in our system are preemptive and independent, or they are changed into independent tasks, so the tasks representing the sporadic servers are also independent and preemptive. So, if the total size U of all the sporadic servers (i.e., the total utilization factor of all the tasks representing the servers) in processing core is less than or equal to $k(2^{1/k}-1)$, all the tasks representing the servers are schedulable according to RM algorithm in processing core. Then all the sporadic servers are schedulable according to RM algorithm in processing core.

B. Schedulability Condition of Real-time Task with EDF Algorithm

Theorem 2: *Under the condition that all the sporadic servers are schedulable and all the tasks do not share critical resources, a new task τ_i with an execution time e_i, a relative deadline d_i and an EDF scheduling policy is released. If*

$$U_t + e_i/d_i \le U_{EDF},$$

*the task τ_i is schedulable in a processing core .Where U_t is the current total process*or utilization factor *of all the tasks in the sporadic server with EDF algorithm in the processing core, and U_{EDF} is the server size of sporadic server with EDF in the processing core.*

Proof: Under the condition that all the sporadic servers are schedulable and all the tasks do not share critical resources, since $U_t + e_i/d_i \le U_{EDF}$, and all the tasks are independent, evidently, according to theorem 7 in [19] the new task τ_i is schedulable in the system with a slow processor with speed $\sigma = U_{EDF}$. Therefore, the task is schedulable in the sporadic server S_{EDF} with size σ in processing core and can be allocated to the processing core.

Corollary 1 *Under the condition that all the sporadic servers are schedulable and all the tasks maybe share some critical resources, a new task τ_i with an execution time e_i, a relative deadline d_i, the worst-case blocking time B_i and an EDF scheduling policy is released. If*

$$U_t + e_i/d_i + B_i/d_i \le U_{EDF},$$

*the task τ_i is schedulable in a processing core. Where U_t is the current total process*or utilization factor *of all the tasks in the sporadic server with EDF algorithm in the processing core and U_{EDF} is the server size of sporadic server with EDF in the processing core.*

Proof: Since the worse-case blocking time of τ_i is B_i, we can take task τ_i as a new task τ_k with an execution time $e_i + B_i$, a relative deadline d_i, an EDF scheduling policy and the worst-case blocking time 0, i.e., the execution of the new task τ_k does not depend on the presence of shared resources. Since there is $U_t + e_i/d_i + B_i/d_i \le U_{EDF}$, evidently, the new task τ_k is schedulable in the system according to **theorem 1**. So the task τ_i is also schedulable in the processing core.

C. Schedulability Condition of Real-time Task with RM Algorithm

Theorem 3: *Under the condition that all the sporadic servers are schedulable and all the tasks do not share critical resources, a new task τ_i with an execution time e_i, a period p_i and a RM scheduling policy is released. If*

$$U_t + e_i/p_i \le (k+1)(2^{1/(k+1)}-1)U_{RM},$$

*the task τ_i is schedulable in a processing core .Where U_t is the current total process*or utilization factor *of all the tasks in the sporadic server with RM algorithm in the processing core, and U_{RM} is the server size of sporadic server with RM, and k is the current number of the tasks with RM algorithm in sporadic server S_{RM} in the processing core.*

Proof: Under the condition that all the sporadic servers are schedulable and all the tasks do not share critical resources, since $U_t + e_i/p_i \le (k+1)(2^{1/(k+1)}-1)U_{RM}$, and all the tasks are independent, evidently, according to theorem 4 in [19] the new task τ_i is schedulable in the system with a slow processor with speed $\sigma = U_{RM}$. Therefore, the task is schedulable in the sporadic server S_{RM} with size σ in processing core and can be allocated to the processing core.

Corollary 2: *Under the condition that all the sporadic servers are schedulable and all the tasks maybe share some critical resources, a new task τ_i with an execution time e_i, a period p_i, the worst-case blocking time B_i and a RM scheduling policy is released. If*

$$U_t + e_i/p_i + B_i/p_i \le (k+1)(2^{1/(k+1)}-1)U_{RM},$$

*the task τ_i is schedulable in a processing core .Where U_t is the current total process*or utilization factor *of all the tasks in the sporadic server with RM algorithm in the processing core, and U_{RM} is the server size of sporadic server with RM, and k is the current number of the tasks*

© 2010 ACADEMY PUBLISHER

with RM algorithm in sporadic server S_{RM} in the processing core.

Proof: Since the worse-case blocking time of τ_i is B_i, we can take task τ_i as a new task τ_k with an execution time e_i+B_i, a period p_i, a RM scheduling policy and the worst-case blocking time 0, i.e., the execution of the new task τ_k does not depend on the presence of shared resources. Since there is $U_t+e_i/p_i+B_i/p_i \leq (k+1)(2^{1/(k+1)}-1)U_{RM}$, evidently, the new task τ_k is schedulable in the system according to **theorem 1**. So the task τ_i is also schedulable in the processing core.

D. Schedulability Condition of Soft Real-time Task with SD Algorithm

Most of the SD scheduling algorithms are based on the notion of General Processor Sharing (GPS) [20-23]. In our architecture, we use a sporadic server to execute the soft real-time tasks with SD algorithm. Suppose a GPS server executes at a rate U_{SD} (which is less than or equal to one), and each task t_i has a reservation ratio u_i which is a positive real number. Each task t_i is guaranteed to be served at a rate of

$$g_i = \frac{u_i}{\sum_j u_j} U_{SD}$$

independent of the actual workloads of other tasks. In other words, the guaranteed CPU service rate g_i for taskt_i will not be affected by the actual behavior of any t_j, i≠ j, i.e., a real-time task can meet its deadline as long as its actual workload does not exceed its reserved rate, i.e. $u_i \leq g_i$.

Theorem 4: *Under the condition that all the sporadic servers are schedulable, a task τ_i with SD algorithm is released, if $u+U^{SD} \leq U_{SD}$, where u is the processor utilization factor of task τ_i, U^{SD} and U_{SD} denote the current total processor utilization factors of all the tasks in sporadic server S_{SD} and server size U_{SD} in processing core respectively, the task is schedulable in processing core and its QoS (quality of service) can be guaranteed well.*

Proof: Since all the sporadic servers are schedulable, the server bandwidth of sporadic server S_{SD} can be guaranteed, and we have

$$u+U^{SD} \leq U_{SD}$$

$$1 \leq \frac{1}{u+U^{SD}} U_{SD}$$

$$u \leq \frac{u}{u+U^{SD}} U_{SD} = g_i$$

Therefore, task τ_i is schedulable in the sporadic server S_{SD} with size U_{SD} in processing core and its QoS can be guaranteed well.

VI. Performance Evaluation

A. Load Balancing

First, we did an experiment to evaluate our load balancing allocation algorithm. In our experiment, we used 100 independent tasks, including 60 hard real-time periodic tasks, 20 soft real-time aperiodic tasks and 20 non-real-time tasks. These tasks ran on our real-time system with 2.4G dual-core processor and 2G RAM. The experimental result is shown as Figure 5.

From the experimental result we can see that the workload of processing core A is close to that of processing core B at most of the time when the system ran. Our algorithm could balance the workload in the system efficiently.

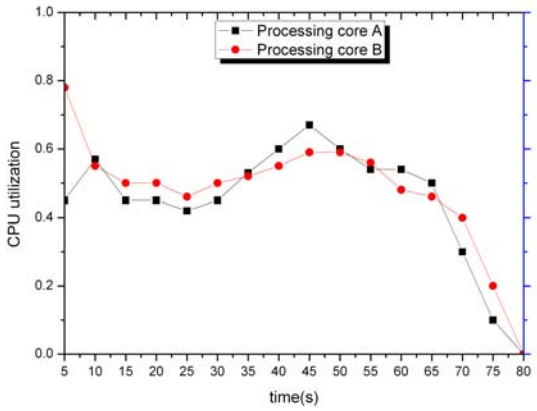


Figure 5. CPU utilization of core A and core B

B. Real-time Criterion

One common criterion to evaluate the performance of a real-time scheduling framework is to measure the overhead used by the scheduler. When comparing two different scheduler implementations, we usually compare the time used to make scheduling decisions.

For real-time systems, another important performance criterion is the task response time which includes interrupt delay, context switch time, scheduling delay and others. Different implementations of the same scheduling algorithm may get different scheduling delays. The scheduling delay in our hybrid scheduling scheme is the time from the occurrence of a scheduling chance to before the context switch for this scheduling point.

Another important performance criterion is the deadline missing rate, which is defined as the ratio of the number of the real-time task instances having missed their deadlines to the number of all real-time task instances. Each sporadic task is a task instance. Each period of a periodic task is a task instance.

For the purpose of comparison with our hybrid scheduling scheme, we also implemented a scheduler based on pure RM and a scheduler based on pure EDF. We did three experiments (using one of the three schemes respectively) in our real-time system with 2.4G dual-core processor and 2G RAM.

In these experiments, we used 100 independent and preemptive hard real-time periodic tasks. In the experiments, half of the tasks use EDF algorithm, and the other half use RM algorithm. The execution time of each task was known a priori. The deadline of periodic task was same as the end of its period. The three experiments used the same way in that these tasks were submitted, and the interval between the time when a task was submitted

© 2010 ACADEMY PUBLISHER

and the time when its previous/next task was submitted was a second. In all experiments presented in this section, data points were collected during the second after one task was submitted. Additionally, the schedulability process was removed in the three experiments.

Figure 6 shows the variance of average scheduling latency with the number of the tasks running in the system. The result indicates that the scheduler with the hybrid scheduling scheme has lower overheads than EDF and higher overheads than RM under the same environment and workload.

Figure 7 shows the variance of deadline missing rate with the number of the tasks running in the system. From Figure 7, we can see that, under the same environment and workload, the deadline missing rate of the real-time systems with pure RM scheduling scheme is higher than the other two schemes, and the lowest is pure EDF scheme.

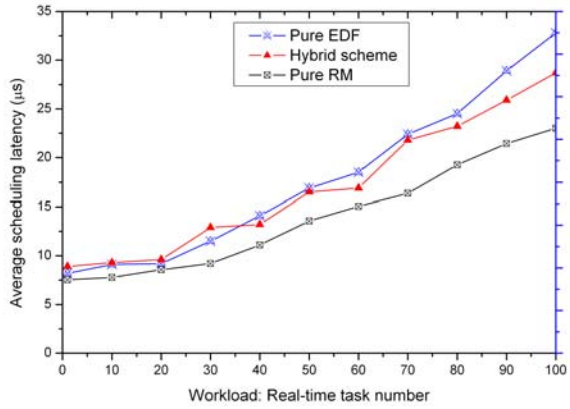


Figure 6. Variance of average scheduling latency

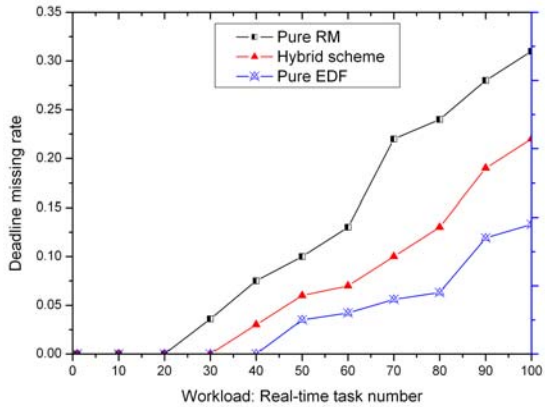


Figure 7. Variance of deadline missing rate

On the whole, the performance of the hybrid scheduling scheme presented in this paper is higher than the pure EDF and RM schemes. Additionally, the hybrid scheme is suitable for jointly scheduling hard, soft and non-real-time tasks in real-time systems but the other two schemes cannot. To get steady testing data, we only used periodic hard real-time tasks in the experiments. Periodic and sporadic hard real-time tasks may coexist in the practical systems. EDF algorithm is the most suitable algorithm that executes sporadic hard tasks, and RM algorithm is the most suitable algorithm executing periodic hard tasks. So, if we use EDF and RM to execute sporadic hard tasks and periodic hard tasks respectively in the practical systems, the performance will be improved a little. Further, for many embedded systems, the flexibility and the reconfigurability of the kernel are more important than its scheduling overheads. The hybrid scheme can be used to implement different real time systems with different goals (such as hard, soft or hybrid real-time systems) and can provide the real-time services with different QoS to the different task modes by adjusting every sporadic server's size. For example, if we increase the size of the sporadic server with hard real-time scheduling policy and decrease the size of the sporadic server with soft real-time scheduling policy, the hard real-time power of the system will be improved, in reverse, the soft real-time power of the system will be improved. The hybrid approach also simplifies the schedulability analysis and validates the schedulability of the tasks with one scheduling policy independently of the tasks with another scheduling policy.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes a hybrid scheduling method for real-time systems on the homogeneous multi-core architectures, which allows the real-time applications to run with non-real-time applications concurrently and supports the parallelism between the tasks within an application efficiently. The method can exploit the parallelism of the multi-core architectures efficiently. In this method, the tasks in one application can run on different processing core in parallel. In addition, each processing core uses a two-level hierarchical scheduling scheme which separates scheduling mechanism from scheduling policy. The schedulability test of the tasks with one scheduling policy can be validated independently of the tasks with other scheduling policies. The scheme can not only meet the timing constraints of the real-time tasks, but also improve the responsiveness of the non-real-time tasks.

Integration of a multi-core chip drives production yields down and they are more difficult to manage thermally than lower-density single-chip designs. In addition, two or more processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage. If a single core is close to being memory bandwidth limited, going to dual-core might only give 30% to 70% improvement. In the future work, we will consider the memory scheduling problem for the multi-core architectures. In addition, this paper only discusses the homogeneous multi-core architectures. Future research will also investigate the heterogeneous multi-core architectures.

© 2010 ACADEMY PUBLISHER

REFERENCES

[1] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol.38, pp. 11-13, 2005.

[2] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California," *IEEE Solid-State Circuits Newsletter*, vol.12, pp. 19-20, 2007.

[3] J. H. Anderson, J. M. Calandrino and U. C. Devi, "Real-Time Scheduling on Multicore Platforms," in *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pp. 179-190.

[4] J. M. Calandrino, D. Baumberger, L. Tong, S. Hahn and J. H. Anderson, "Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms," in *Proc. 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, pp. 101-112.

[5] H. A. James and M. C. John, "Parallel Real-Time Task Scheduling on Multicore Platforms," in *Proc. 27th IEEE International Real-Time Systems Symposium, 2006 (RTSS '06)*, pp. 89-100.

[6] H. Leontyev and J. H. Anderson, "Tardiness Bounds for EDF Scheduling on Multi-Speed Multicore Platforms," in *Proc. 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pp. 103-110.

[7] M. Cirinei, E. Bini, G. Lipari and A. Ferrari, "A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors," in *Proc. IEEE International Parallel and Distributed Processing Symposium, 2007 (IPDPS 2007)*, pp. 1-8.

[8] J. M. Calandrino, J. H. Anderson and D. P. Baumberger, "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms," in *Proc. 19th Euromicro Conference on Real-Time Systems, 2007 (ECRTS '07)*, pp. 247-258.

[9] S. Euiseong, J. Jinkyu, P. Seonyeong and L. Joonwon, "Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol.19, pp. 1540-1552, 2008.

[10] C. Shih-Ying and H. Chih-Wen, "Optimal Dynamic-Priority Real-Time Scheduling Algorithms for Uniform Multiprocessors," in *Proc. 29th IEEE Real-Time Systems Symposium, 2008 (RTSS'08)*, pp. 147-156.

[11] B. B. Brandenburg, J. M. Calandrino and J. H. Anderson, "On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study," in *Proc. 29th IEEE Real-Time Systems Symposium, 2008 (RTSS'08)*, pp. 157-169.

[12] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit and J. Duato, "A simple power-aware scheduling for multicore systems when running real-time applications," in *Proc. IEEE International Parallel and Distributed Processing Symposium, 2008 (IPDPS 2008)*, pp. 1-7.

[13] J. M. Calandrino and J. H. Anderson, "Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study," in *Proc. 20th Euromicro Conference on Real-Time Systems, 2008 (ECRTS '08)*, pp. 299-308.

[14] L. Liang-Teh, C. Huang-Yuan and C. Shu-Wei, "A Hybrid Task Scheduling for Multi-Core Platform," in *Proc. Second International Conference on Future Generation Communication and Networking Symposia, 2008 (FGCNS '08)*, pp. 40-45.

[15] B. Sprunt, L. Sha and J. Lehoczky, "Aperiodic task scheduling for Hard-Real-Time systems," *Real-Time Systems*, vol.1, pp. 27-60, 1989.

[16] Z. Deng and J. W. S. Liu, "Scheduling real-time applications in an open environment," in *Proc. The 18th IEEE Real-Time Systems Symposium, 1997*, pp. 308-319.

[17] K. Tei-Wei and L. Ching-Hui, "A fixed-priority-driven open environment for real-time applications," in *Proc. The 20th IEEE Real-Time Systems Symposium, 1999*, pp. 256-267.

[18] K. Tei-Wei, L. Kwei-Jay and W. Yu-Chung, "An open real-time environment for parallel and distributed systems," in *Proc. 20th International Conference on Distributed Computing Systems, 2000*, pp. 206-213.

[19] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment ," *J. ACM* , vol.20 , pp. 46-61, 1973.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proc. 17th IEEE Real-Time Systems Symposium, 1996*, pp. 288-299.

[21] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case ," *IEEE/ACM Trans. Netw.* , vol.1 , pp. 344-357, 1993.

[22] K. Tei-Wei, Y. Wang-Ru and L. Kwei-Jay, "EGPS: a class of real-time scheduling algorithms based on processor sharing," in *Proc. 10th Euromicro Workshop on Real-Time Systems, 1998*, pp. 27-34.

[23] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. The 19th IEEE Real-Time Systems Symposium, 1998*, pp. 4-13.

Pengliu Tan was born on October 27, 1975, in Xianning, Hubei Province, China. He received a Ph.D. in Computer Architecture from the Huazhong University of Science and Technology in 2008.

He is currently a lecturer at the school of software, Nanchang Hangkong University, China. His research interests include real-time systems, task scheduling, wireless sensor network, Cyber-Physical Systems.

Jian Shu was born on May 25, 1964, in Nanchang, Jiangxi Province, China. He received a Ms. in Computer Networks from Northwestern Polythecnical University in 1990.

He is currently a professor at the school of software, Nanchang Hangkong University, China. His research interests include wireless sensor network, embeded system and software engineering

Zhenhua Wu was born on November 1, 1977, in Poyang County, Jiangxi Province, China. He received a Ph.D. in Computer Architecture from the Huazhong University of Science and Technology in 2006.

He is currently a lecturer at the school of software, Nanchang Hangkong University, China. His research interests include wireless sensor network, Intelligent information processing and pattern recognition

© 2010 ACADEMY PUBLISHER