International Journal of Computer Applications (0975 – 8887)
Volume 65– No.5, March 2013

An Approach for Multi-Core Real Time Parallel Processing

Shyamal G. Mundada
Department of Computer Science Engineering
Ramdeobaba College Of Engineering and Management
Nagpur, India

M.B.Chandak
Head & Associate Professor, Department of Computer Science Engineering
Ramdeobaba College Of Engineering and Management
Nagpur, India

ABSTRACT

Multi-core architectures, which have multiple processing units on a single chip, are widely used as a way to achieve higher processor performance. They have potential to deliver increased performance over single-core processors. Multi-core processors have become mainstream in processor design. In multiprocessing, only inter task parallelism can be achieved. But, computation-intensive real-time systems must exploit intra-task parallelism to take full advantage of multi-core processing. In this paper, the problem of scheduling periodic parallel tasks with implicit deadlines on multi-core processors is addressed. A task decomposition method that decomposes each parallel task into a set of sequential tasks is discussed. In this paper, a general model for deterministic parallel tasks, where a task is represented as a DAG with different nodes having different execution requirements is discussed. First, a DAG generation method for the tasks is discussed and secondly, task decomposition that splits a DAG into sequential tasks is discussed.

General Terms

Real Time Scheduling

Keywords

Multi-core Processing, Real Time Scheduling, Directed Acyclic Graph.

1. INTRODUCTION

In Recent Years, Multi-core Processor Technology has improved dramatically as chip manufacturers try to boost performance while minimizing power consumption. This development has shifted the scaling trends from increasing processor clock frequencies to increasing the number of cores per processor. For example, Intel has recently put 80 cores in a Teraops Research Chip (Intel, 2007) with a view to making it generally available, and ClearSpeed has developed a 96-core processor (ClearSpeed, 2008). While hardware technology is moving at a rapid pace, software and programming models have failed to keep pace. For example, Intel (2007) has set a time frame of 5 years to make their 80-core processor generally available due to the inability of current operating systems and software to exploit the benefits of multi-core processors. As multi-core processors continue to scale, they provide an opportunity for performing more complex and computation-intensive tasks in real-time. However, to take full advantage of multi-core processing, these systems must exploit intra-task parallelism, where parallelizable real-time tasks can utilize multiple cores at the same time. By exploiting intra-task parallelism, multi-core processors can achieve significant real-time performance improvement over traditional single-core processors for many computation-intensive real-time applications such as video surveillance, radar tracking, and hybrid real-time structural testing (Huang et al., 2010) where the performance limitations of traditional single-core processors have been a major hurdle.

Many models of parallelism have been used in programming languages and Application Program Interfaces, but few of them have been studied in real-time systems. An existing technique for real time scheduling considers a synchronous task model, where each parallel task consists of a series of sequential or parallel segments. This model is treated as synchronous, since all the threads of a parallel segment must finish before the next segment starts, creating a synchronization point. However, the task model is restrictive in that, for every task, all the segments have an equal number of parallel threads, and the execution requirements of all threads in a segment are equal. Most importantly, in this task model, the number of threads in every segment is no greater than the total number of processor cores. The restrictions on the task model make the solutions unsuitable for many real time applications that often employ different numbers of threads in different segments of computation.

One more technique for real time scheduling on multi-core processors considers a more general synchronous task model. Here also, tasks contain segments where the threads of each segment synchronize at its end. However, in contrast to the restrictive task model addressed, for any task in this model, each segment can contain an arbitrary number of parallel threads. That is, different segments of the same parallel task can contain different numbers of threads, and segments can contain more threads than the number of processor cores. The execution requirements of the threads in any segment can vary. This model is more portable, since the same task can be executed on machines with small as well as large numbers of cores.

In this paper, a general task model is considered, where tasks are represented by general DAGs where threads (nodes) can have *arbitrary* execution requirements. The paper is organized as follows. Section 2 reviews related work. Section 3 describes the task model and DAG generation. Section 4 presents the task decomposition. Section 5 offers conclusions.

2. RELATED WORK

There has been a substantial amount of work on traditional multiprocessor real-time scheduling focused on sequential tasks [4]. Some work has addressed scheduling for parallel tasks [9]–[10], but it does not consider task deadlines. Soft Real time scheduling (where the goal is to meet a certain subset of deadlines based on application-specific criteria) has been studied for various parallel task models and for various optimization criteria [11]–[13].Hard real-time scheduling (where the goal is to meet all task deadlines) is intractable for most cases of parallel tasks without resource augmentation [14].

The problem of scheduling implicit-deadline periodic task sets on multiprocessor systems under fork-join structure was introduced in [6]. A task stretch transform was defined which is to be performed by OS scheduler. Partitioned fixed priority scheduling algorithm was used for scheduling periodic fork-join task set. The problem of scheduling periodic parallel task by considering intra task parallelism was addressed in [5]. A synchronous task model was considered .A new task decomposition technique was introduced to transform each parallel task into a set of sequential tasks. These tasks were then scheduled using global EDF and partitioned deadline monotonic scheduling. All parallel segments in a task have an equal number of threads which cannot exceed the number of processor cores. It transforms every thread to a subtask, and proves a resource augmentation bound of 3:42 under partitioned Deadline Monotonic (DM) scheduling. For the synchronous model with arbitrary numbers of threads in segments, our earlier work in [6] proves a resource augmentation bound of 4 and 5 for global EDF and partitioned DM scheduling, respectively. For the unit-node DAG model where each node has unit execution requirement, this approach converts each task to a synchronous task, and then applies the same approach. A scheduling method, 'spread-cognizant' was proposed in [13] that decreases average and maximum spreads in global EDF scheduling algorithms. The scheduling mechanism for individual threads of multithreaded real time tasks was discussed.

In this paper, a more general model of deterministic parallel real-time tasks is considered where each task is modeled as a DAG, and different nodes of the DAG may have different execution requirements.

3. PARALLEL TASK MODEL

Parallel implicit-deadline real time tasks are considered for scheduling, where the deadline of a task equals to their period, and each task of this model is represented by a directed acyclic graph (DAG), which is a collection of subtasks and directed edges, which represents the execution flow of the task and the precedence constraints between the subtasks. Precedence constraint means that each node can start its execution when all of its predecessors have finished theirs.

A set of 'n' periodic parallel tasks is considered to be scheduled on a multi-core platform consisting of *m* identical cores. The task set is represented by $\tau_i = \{ \tau_1, \tau_2, \ldots, \tau_n \}$.Each task τ_i , $1 \le i \le n$, is represented as a Directed Acyclic Graph(DAG),where the nodes stand for different execution requirements, and the edges represent dependencies between the nodes. A node in τ_i is denoted by W_i^j; $1 \le j \le n_i$, with ni being the total number of nodes in τ_i . The execution requirement of node W_i^j is denoted by E_i^j. A directed edge from node W_i^j to node W_i^k, denoted as $W_i^j \rightarrow W_i^k$, implies that the execution of W_i^k cannot start unless W_i^j has finished execution. W_i^j in this case, is called a parent of W_i^k, while W_i^k is its child. A node may have 0 or more parents or children. A node can start execution only after all of its parents have finished execution. Figure 1 shows a task τ_i with $n_i = 6$ nodes.

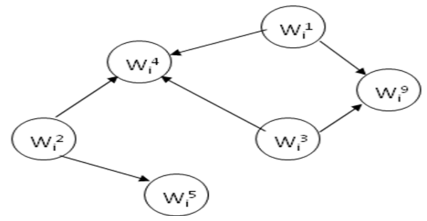

Fig 1 A parallel task τ_i represented as a DAG

The total execution requirement of τ_i is the sum of the execution requirements of all of its nodes, and is denoted by C_i (time units). The period of task τ_i is denoted by P_i. The deadline D_i of each task τ_i is considered implicit, i.e., $D_i = P_i$. Task set τ is said to be schedulable by algorithm A, if A can schedule τ such that every $\tau_i \in \tau$ can meet deadline D_i. Representing jobs and their precedence constraints as a DAG is very convenient. It gives some interesting information about the problem like the degree of parallelism that application can attain the minimum amount of time required by the application (the critical path of the graph), etc. These properties can be used for the development of heuristics to solve scheduling problems.

There are various graph generation algorithms which can be used in the validation of scheduling algorithms. A random graph is obtained by starting with a set of *n* vertices and adding edges between them at random. Different random graph models produce different probability distributions on graphs. Most commonly used methods are the Erd˝os-Rényi methods. There are two methods G(n,p) and G(n,M).We consider G(n,p) for graph generation.

It is the most intuitive and most widely utilized graph generation method. For given n number of vertices, the G (n,p) method generates a graph where each element of the (n,2) possible edges is present with independent probability p. Erdos and Renyi first defined this method for non-oriented graphs, but it is easy to adapt this for DAG generation:

Algorithm: G (n,p) method

```
Require: n ε N; p ε R.
Ensure: a graph with n nodes.
Let M be an adjacency matrix n x n initialized as the
zero matrix.
for all i = 1 to n do
        for all j = 1 to i do
        if Random() < p then
                M[i][j] = 1
        else
                M[i][j] = 0
        return the graph represented by M.
```

4. TASK DECOMPOSITION

Scheduling parallel tasks by decomposing them into sequential subtasks is considered here. This strategy allows leveraging existing schedulability analysis for multiprocessor scheduling (both preemptive and non-preemptive). In this section, the decomposition of a parallel task under general DAG model is presented. The method decomposes a task into nodes. Thus, each node of a task becomes a sequential subtask with execution requirement equal to the execution requirement of the node. All nodes of a DAG are assigned appropriate deadlines and release offsets such that when they execute as individual subtasks all dependencies among them in the DAG (i.e., in the original task) are preserved. Thus, an implicit deadline DAG is decomposed into a set of constrained deadline sequential subtasks with each subtask corresponding to a node of the DAG.

4.1 Related Terms

4.1.1 *Execution Requirement of task*

C_i: It is the sum of the execution requirements of all nodes in T_i.

$$Ci = \sum_{j=1}^{ni} Eij$$

$C_{i,v}$: Maximum execution time of task T_i on a multi-core platform where each processor has speed v.

$$Ci,v = 1/v \sum_{j=1}^{ni} Eij$$

4.1.2 *Critical path length*

P_i: It is the Sum of the execution requirements of the nodes on a critical path. Denotes Minimum execution time of task τ_i .

$$Ti \geq Pi$$

4.1.3 *Utilization of task*

$$Ui = {}^{Ci}/_{Ti}$$

4.1.4 *Density of task*

$$\delta i = {}^{Ci}/_{Di}$$

4.2 Decomposition Technique

In the decomposition, each node of a task becomes an individual sequential subtask with its own execution requirement and an assigned constrained deadline. To preserve the dependencies in the original DAG, each node is assigned a release offset. Since a node cannot start execution until all of its parents finish, its release offset is equal to the maximum sum of the release offset and deadline among its parents. That is, a node starts after its *latest* parent finishes. The (relative) deadlines of the nodes are assigned by distributing the available slack of the task. The slack for each task considering a multi-core platform where each processor core has speed 2 is calculated. The *slack* for task τ_i , denoted by *Li*, is defined as the difference between its deadline and its critical path length on 2-speed processor cores.

$$L_i = D_i - P_{i,2} = T_i - P_{i,2} = T_i - P_{i/2}$$

For task τ_i, the deadline and the offset assigned to node W_i^j are denoted by D_i^j and Φ_i^j, respectively. Once appropriate values of D_i^j and Φ_i^j are determined for each node W_i^j (respecting the dependencies in the DAG), task τ_i is decomposed into nodes. Upon decomposition, the dependencies in the DAG need not be considered, and each node can execute as a traditional multiprocessor task. Hence, the decomposition technique for τi boils down to determine D_i^j and Φ_i^j for each node W_i^j.

Consider DAG given in Figure 1, here we assign execution requirement to each node as E_i^1=4, E_i^2=2, E_i^3=4, E_i^4=5 and E_i^5=3. First, DAG *τi is* represented as a *timing diagram* $\tau_i^{original}$. Specifically, $\tau_i^{original}$ indicates the earliest start time and the earliest finishing time of each node. For any node W_i^j that has no parents, the *earliest start time* and the *earliest finishing time* are 0 and E_i^j, respectively. For every other node W_i^j, the *earliest start time* is the latest finishing time among its parents, and the *earliest finishing time* is E_i^j time units after that. For example, in τi of Figure 1, nodes W_i^1,W_i^2, and W_i^3 can start execution at time 0, and their earliest finishing times are 4, 2, and 4,respectively. Node W_i^4 can start after W_i^1 and W_i^2 complete, and finish after 5 time units at its earliest, and so on. Thus, Figure 2(a) shows $\tau_i^{original}$ of the DAG τi of Figure 1.

The calculation of D_i^j and Φ_i^j for each node W_i^j involves the following two steps. In Step 1, for each node, distribution of slack among different parts of the node is done. In Step 2, the total slack assigned to different parts of the node is assigned as the node's slack.

Step 1 (slack distribution): In DAG τ_i, a node can execute with different numbers of nodes in parallel at different time. Such a degree of parallelism can be approximated based on $\tau_i^{original}$. For example, in Figure 2(a), node W_i^5 executes with W_i^1 and W_i^3 in parallel for the first 2 time units, and then executes with W_i^4 in parallel for the next time unit. In this way, identification of the degrees of parallelism at different parts of each node is carried out. Intuitively, the parts of node which execute with a large number of nodes in parallel require more slack. Therefore, different parts of a node are assigned different amounts of slack considering their degrees of parallelism and execution requirements. Later, the sum of slack of all parts of a node is assigned to the node itself. To identify the degree of parallelism for different portions of a node based on τ_i^{∞}, assignment of slack to a node in different (consecutive) segments is done. In different segments of a node, the task may have different degrees of parallelism. In τ_i^{∞}, starting from the left, a vertical line at every time instant is drawn where a node starts or ends (as shown in Figure 2(b)). This is done using a breadth-first search over the DAG. The vertical lines now split τ_i^{∞} into segments. For example, in Figure 2(b), τ_i is split into 2 segments (numbered in increasing order from left to right).Once τ_i^{∞} is split into segments, each segment consists of an equal amount of execution by the nodes that lie in the segment. Parts of different nodes in the same segment can now be thought of *threads* that can run in parallel, and the threads in a segment can start only after those in the preceding one finish. Such a model is thus similar to the synchronous task model used in [6]. This model is denoted by τ_i^{syn}. Assignment of slack to the segments is done, and finally addition of all slack assigned to different segments of a node is carried out to calculate its overall slack.

Distribution of slack is done among the nodes based on the number of threads and execution requirement of the segments where a node lies in τ_i^{syn}. For every *j*-th segment of τ_i^{syn}, we calculate a value d_i^j, called an *intermediate subdeadline* .That is, each thread in the segment gets this "extra time" beyond its execution time.

Step 2 (deadline and offset calculation): Intermediate sub deadlines to (the threads of) each segment of τ_i^{syn} are assigned in Step 1. Since a node may be split into multiple (consecutive) segments in τ_i^{syn}, now we have to remove all intermediate sub deadlines of a node. Addition of all intermediate sub deadlines of a node is done, and the total is assigned as the node's deadline.

Now let a node W_i^j of τ_i belong to segments k to r ($1 \le k \le r \le s_i$) in τ_i^{syn}. Therefore, the deadline D_i^j of node W_i^j is calculated as follows.

$D_i^j = d_i^k + d_i^{k+1} + \dots\dots\dots + d_i^r$

The execution requirement E_i^j of node W_i^j is

$E_i^j = e_i^k + e_i^{k+1} + \dots\dots\dots + e_i^r$

Node W_i^j cannot start until all of its parents complete. Hence, its release offset Φ_i^j is determined as follows.

$$\Phi_i^j = \begin{cases} 0\,; if\ W_i^j\ has\ no\ parent \\ \max\{\emptyset_i^l + D_i^l \mid W_i^l\ is\ a\ parent\ of\ W_i^j\}\,; otherwise \end{cases}$$

Appropriate deadline D_i^j and release offset Φ_i^j to each node W_i^j of τ_i are assigned. The DAG τ_i is now decomposed into nodes. Each node W_i^j is now an individual (sequential) multiprocessor subtask with an execution requirement E_i^j, a constrained deadline D_i^j, and a release offset Φ_i^j.

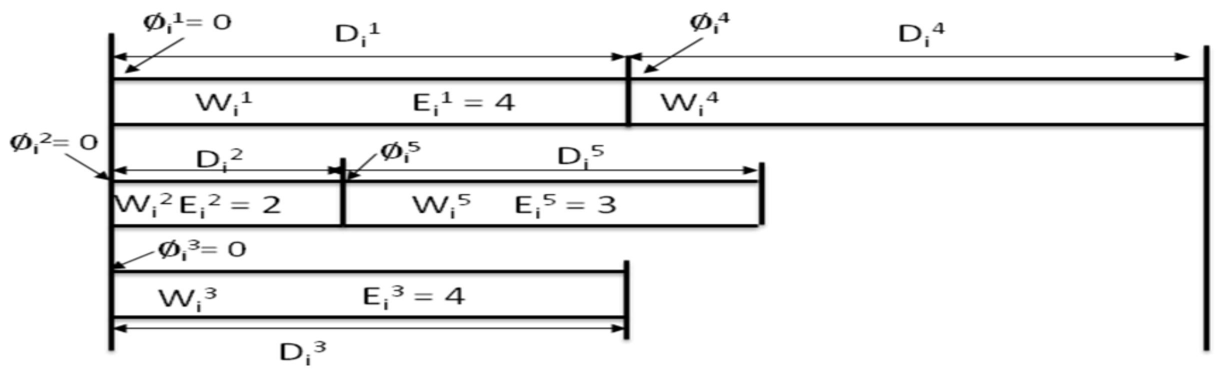

Fig 2 (a) $\tau_i^{original}$: Timing diagram for DAG τ_i

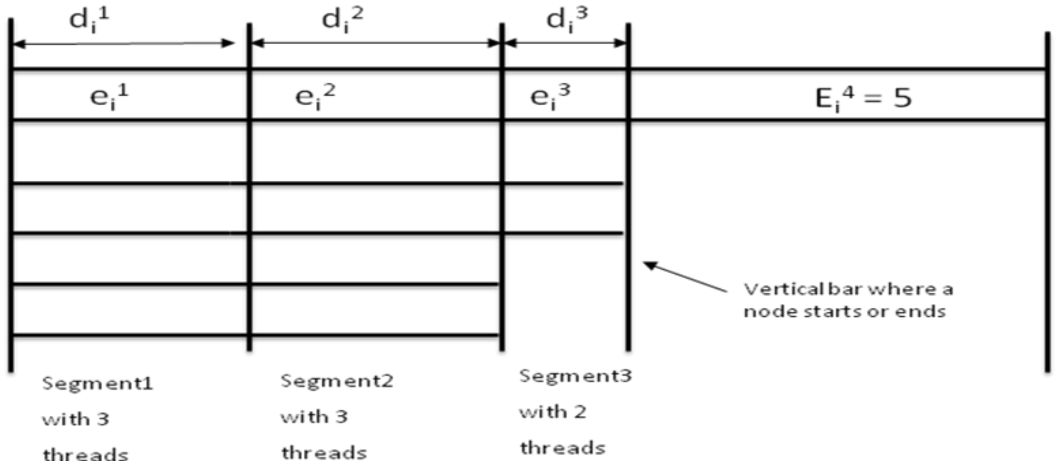

Fig 2 (b) Slack distributions in τ_i^{syn}

5. CONCLUSIONS

With the advent of the era of multi-core computing, real time scheduling of parallel tasks is crucial for real-time applications to exploit the power of multi-core processors. While recent research on real-time scheduling of parallel tasks has shown promise, the efficacy of existing approaches is limited by their restrictive parallel task models. To overcome these limitations, in this paper generalized parallel task model for real-time scheduling is presented. A general synchronous parallel task model is considered where each task consists of segments, each having an arbitrary number of parallel threads. Then a novel task decomposition algorithm is discussed which decomposes each task into a set of tasks which can be scheduled on multiple cores.

6. REFERENCES

[1] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” ACM Comp. Surv., vol. 43, pp. 35:1–44, 2011.

[2] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” in RTSS ’11.

[3] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Scheduling parallel realtime tasks on multi-core processors,” in RTSS ’10.

[4] “OpenMP,” http://openmp.org.

[5] “Intel CilkPlus,” http://software.intel.com/en-us/articles/intel-cilk-plus.

[6] X. Deng, N. Gu, T. Brecht, and K. Lu, “Preemptive scheduling of parallel jobs on multiprocessors,” in SODA ’96.

[7] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, “Adaptive task scheduling with parallelism feedback,” in PPoPP ’06.

[8] J. M. Calandrino and J. H. Anderson, “On the design and implementation of a cache-aware multicore real-time scheduler,” in ECRTS ’09.

[9] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, “A hybrid real-time scheduling approach for large-scale multicore platforms,” in ECRTS ’07.

[10] J. H. Anderson and J. M. Calandrino, “Parallel real-time task scheduling on multicore platforms,” in RTSS ’06.

[11] C.-C. Han and K.-J. Lin, “Scheduling parallelizable jobs on multiprocessors,” in RTSS ’89.

[12] S. Baruah, “Techniques for multiprocessor global schedulability analysis,”in RTSS ’07.

[13] J. Goossens, S. Funk, and S. Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors,” Real-Time Syst., vol. 25, no.2-3, pp. 187–205, 2003.

[14] S. Baruah, “The non-preemptive scheduling of periodic tasks upon multiprocessors,” Real-Time Syst., vol. 32, pp. 9–20, 2006.

[15] Jos´e Carlos Fonseca, Lu´ıs Nogueira, Cl´audio Maia, and Lu´ıs Miguel Pinho,”Real-Time Scheduling of Parallel Tasks in the Linux Kernel”

[16] D.I. George Amalarethinam and G.J. Joyce Mary, “A new DAG based Dynamic Task Scheduling Algorithm (DYTAS) for Multiprocessor Systems”, *in International Journal of Computer Applications (0975 – 8887) Volume 19– No.8, April 2011*

[17] D.I. George Amalarethinam1 and G.J. Joyce Mary, ”DAGEN - A Tool To Generate Arbitrary Directed Acyclic Graphs Used For Multiprocessor Scheduling”, in International Journal of Research and Reviews in Computer Science (IJRRCS) Vol. 2, No. 3, June 2011

[18] C.-C. Han and K.-J. Lin, “Scheduling parallelizable jobs on multiprocessors,”in RTSS ’89.

[19] K. Jansen, “Scheduling malleable parallel tasks: An asymptotic fullypolynomial time approximation scheme,” Algorithmica, vol. 39, no. 1,pp. 59–81, 2004.

[20] W. Y. Lee and H. Lee, “Optimal scheduling for real-time parallel tasks,”IEICE Trans. Inf. Syst., vol. E89-D, no. 6, pp. 1962–1966, 2006.

[21] S. Collette, L. Cucu, and J. Goossens, “Integrating job parallelism in real-time scheduling theory,” Inf. Process. Lett., vol. 106, no. 5, pp.180–187, 2008.

[22] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, “A new approach for scheduling of parallelizable tasks inreal-time multiprocessor systems,” Real-Time Syst., vol. 15, no. 1, pp. 39–60, 1998

[23] S. Kato and Y. Ishikawa, “Gang EDF scheduling of parallel task systems,” in *RTSS ’09*.

[24] N. Fisher, T. P. Baker, and S. Baruah, “Algorithms for determining the demand-based load of a sporadic task system,” in *TCSA ’06*.

[25] J. Goossens, S. Funk, and S. Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors,” *Real-Time Syst.*, vol. 25, no. 2-3, pp. 187–205, 2003.

[26] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, Frédéric Wagner, ”Random graph generation for scheduling simulations” in simutools’10

[27] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” in *SPAA ’98.*

[28] Ricardo Garibay-Martínez, Luis Lino Ferreira, Luis Miguel Pinho,”A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems “

[29] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comp. Surv., vol. 43, pp. 35:1–44, 2011.*