Multilevel Static Real-Time Scheduling Algorithms Using Graph Partitioning

Kayhan Erciyes[1] and Zehra Soysert[2]

[1] Izmir Institute of Technology,
Computer Eng. Dept., Urla, Izmir 35340, Turkey
kayhanerciyes@iyte.edu.tr
[2] Ege University International Computer Institute,
35100 Bornova, Izmir, Turkey
soysert@bornova.ege.edu.tr

Abstract. We propose static task allocation algorithms for the periodic tasks of a distributed real-time system. The cyclic task consists of task threads which may communicate and share resources. A graph partitioning process and a thread sequencing algorithm are applied to these threads to yield local schedules. The exact analysis is then obtained and further refinements are performed if the worst case response time of a task is greater than its deadline.

1 Introduction

Scheduling in real-time systems can be broadly described as static or dynamic. Static scheduling of processes with known release times, deadlines, precedence, and exclusion relations is decribed in [13]. Schedulability tests for *Rate Monotonic* (RM) and *Earliest Deadline First* (EDF) algorithms for cyclic tasks when their deadlines equal their periods are presented in [5]. An exact sheduling test method for RM algorithm [9] is derived in [6]. A method to find the schedulability of a task set when Deadline Monotonic Scheduling is used is described in [1]. The task of scheduling tasks on a multiprocessor/distributed environment is NP-hard [12]. For this reason, various heuristics such as iterative improvement algorithms [7], and the probabilistic optimization as simulated annealing algorithms [10] and genetic algorithms [11] have been proposed. A middleware distributed real-time scheduling method is shown in [2]. The goal of our ongoing work is to investigate the balancing of static load over a distributed real-time system where computation nodes executing real-time kernels are connected by a real-time network that provides bounded mesage delays. Formally, if $T = \{t_1, t_2, ..., t_m\}$ is the set of real-time tasks and $P = \{p_1, p_2, ..., p_n\}$ is the set of processors, we need to derive the mapping function $M : T \rightarrow P$ so that every task meets its deadline. To accomplish this, we consider the periodic tasks of the real-time system consisting of *real-time threads* each with a hard deadline. Threads have interprocess communications and may access shared resources. We sketch the static execution characteristics of these threads which are computation times, interprocess communication patterns and deadlines as a directed graph and use a novel *multilevel graph partitioning* heuristic to divide the graph into n regions. The output of the graph partitioning algorithm are the task thread sets to be executed by each processor. We then sequence these threads for each processor by a *Task Sequencing Algorithm*, providing that the precedence constraints are obeyed and deadlines are met. In the second phase, we apply the exact scheduling analysis to work out the worst case response times $R_{ij}\ for\ j = 1..N_i$ of the individual task components taking the blocking times by lower priority threads in the same or other tasks. If for any task component, R_{ij} is larger than deadline, our partitioning is unsuccesful and we go into refinement phase where we try to move threads from one processor to another to provide $R_{ij} < d_{ij}$.

The rest of the paper is organized as follows. Section 2 provides the computation model and the partitioning algorithm output of which is scheduled by the task sequencing algorithm described in Section 3. The exact analysis and the refinement procedure that is invoked if the exact analysis detects missing deadlines of tasks is described in Section 4. An example of operation is shown in Section 5 and future directions are outlined in the Conclusions Section.

2 Task Graph Partitioning Algorithm

For the hard-real time tasks we assume the following computation model:

- Worst case execution time of each task C_i and its deadline D_i are known in advance and for periodic tasks, deadlines equal periods. $(D_i = P_i)$
- Each task T_i consists of a number of threads t_{ik}, each of which has a computation time c_{ik} and a deadline $d_{ik}\ for\ k = 1..N_i$ where N_i is the count of threads of t_i.
- Task threads have interprocess communication which determine their precedence. If a task thread t_{ij} has to communicate the result of its computation to task thread t_{ik}, we say $t_{ij} \prec t_{ik}$, that is, the execution of t_{ij} precedes the execution of t_{ik}.
- Tasks threads, of the same or different tasks, may access shared resources, therefore may block, competing for these resources. The maximum blocking time b_i for a task and its threads can be determined priori using the *Priority Ceiling Protocol* analysis [8].

The aim of the our partitioning algorithm is to partition the task thread graph into subgraphs so that each task thread meets its deadline and also to provide a partition such that the load (total execution time of task threads) is averaged over all subgraphs. We use a modified multilevel graph partitioning method of [4] where instead of finding maximal matching, the graph is partitioned around fixed centers as in [3]. The task threads t_{ik} of a periodic real-time task t_i can be constructed using a directed graph $G = (V, E, w)$ where V is the set of task threads, E is the set of edges giving interprocess communication between threads and $w : \Re \rightarrow E$ is the set of weights associated with edges. This method has coarsening, partitioning and uncoarsening phases. During the coarsening phase, a set of smaller graphs are obtained from the initial graph $G_i = (V_i, E_i)$ such that $|V_i| > |V_{i+1}|$. When graph G_{i+1} is to be constructed from graph G_i, a maximal matching $M_i \subseteq E_i$ is found and vertices that are incident on both edge of this matching are collapsed. The rules for collapsing are as follows. If $u, v \in V_i$ are collapsed to form vertex $w \in V_{i+1}$, the total weight of vertices u and v become the weight of w, the edges incident on w is set equal to the union of the edges incident on u and v minus the edge (u, v). If there is an edge that is incident on both u and v, then the weight of this edge is set equal to the sum of the weights of these two edges. Vertices that are not incident on any edge of the matching are simply copied over to G_{i+1} [4].

We have adapted the modified version of the method presented in [3] for the real-time case where a directed graph depicts task threads with precedence relations and strict deadlines. The coarsening phase then needed to be modified as follows. When we want to choose a neighbor to collapse around the fixed centers, we first look for any predecessors not assigned to a group yet. If there is more than one predecessor, we apply a combination heuristic $H = W_1 * EDF + W_2 * HEM$ where HEM is the *Heaviest Edge Matching.* In other words, we consider communication costs as well as the earliest deadlines when collapsing. W_1 and W_2 are the weights that can be adjusted. If there are no predecessors, the same process is repeated for the sucessors until the number of vertices in the coarsened graph equals the number of procesors n. Finally, we uncoarsen the coarsened vertices back to get the original graph. This algorithm is depicted in Fig. 1.

```
Procedure Task_Graph_Partition (TTG:Graph(V,E), n:number of processors);
 Begin
   Mark n nodes of the graph as fixed centers;
   While there are nodes to be collapsed
    Apply H to neighbors of centers;
    Add the executon time of neighbors to centers;
    Collapse the chosen neighbors to the centers;
  Partition the coarsened TTG;
  Uncoarsen TTG back to original;
 End.
```

Fig. 1. Task Graph Partitioning Algorithm (TGPA)

Theorem 1. *TGPA performs partitioning of $G(V, E)$ in $O(\lfloor N/n \rfloor)$ steps for each task where $|V| = N$ is an upperbound on the number of task threads and n is the number of processors. The time complexity of the total collapsing of TGPA is $O(mN)$ where m is the number of tasks.*

Proof. The TGPA collapses n nodes using H at each step for each task for $O(\lfloor N/n \rfloor)$ steps which would be $O(N)$ operations in total for one task. Total number of collapsing for all of the tasks is then $O(mN)$.

3 Thread Sequencing Algorithm

The information from the partitioning phase yields the coarse order of the task threads to be executed in one processor such that if t_{ij} and t_{ik} are assigned to the same processor by the partitioning algorithm and $t_{ij} \prec t_{ik}$, then t_{ij} has to be executed before t_{ik}. If these two threads have no precedence constraints, that is $t_{ij} \parallel t_{ik}$, they can be assigned arbitrarily after their release times. The *Task Thread Sequencing Algorithm (TTSA)* that is employed to provide the final schedule for task threads on the processor is shown in Fig.2 where any thread that does not have any predecessors becomes ready to be scheduled.

```
Procedure Thread_Sequence (TTG:Graph(V,E), n:Number of processors);
 Begin
  Task_type=READY for all t_{ik} where n_{ik} = 0; {no predecessors}
  Insert READY tasks into Sched_list;
  While Sched_list is not empty
      Get a task thread t_{ik} from the Sched_list;
      Switch (Event_type)
         case READY :
           Event_type = FINISHED;
           time = time + Finish_time of t_{ik};
           Insert_event into Sched_list;
         case FINISHED :
           For each immediate successor t_{ip} of t_{ik};
              n_{ip} = n_{ip} - 1;
              if n_{ip} = 0
                 Event_type = READY;
                 time=time+finish_time of t_{ik} ;
                 Insert event into Sched_list;
 End.
```

Fig. 2. Thread Sequencing Algorithm (TSA)

4 Exact Analysis and Refinement

In the second phase, we take the partial execution times of tasks per processor as inputs which is the sum of the execution times of all the task threads on the same processor, assuming interprocess communication costs are zero. We then calculate the worst blocking times for these threads as follows. For each thread, we look at the lower priority threads of the same tasks, that is threads that are prior in execution, and all threads of higher priority tasks. From those, we find the semaphores they are using, and choose the semaphores that have a higher or equal ceiling than the task thread under consideration. The maximum critical section time for these semaphores is the blocking time b_{ik} for thread t_{ik}. In this case, a thread cannot be prempted by a lower priority thread of the same task, it can however be preempted by any thread of the higher priority tasks. Based on these assumptions, a worst case response time R_{ij}, that is, the response time of task component t_{ij} on processor j can be calculated as follows:

$$R_{ij}^{n+1} = b_{ij} + c_{ij} + \sum_{\forall l \in hp(i)} (\lceil \frac{R_{ij}^{n}}{T_l} \rceil * c_{lj})$$

where $hp(i)$ is the set of tasks that have higher priorities than t_i, c_{lj} is the computation time of a task component t_{lj} on processor j. We then find $max(R_{ij})$ $for\ j = 1..N_i$. If this is greater than deadline d_i of task t_i then our allocation has failed. In this case, we go into refinement phase where we change the weights of the EDF and HEM heuristics to get different partitionings. This procedure is repeated until a feasible schedule is obtained.

5 An Example of Operation

As an example, consider the task set (T_A, T_B, T_C) with the static properties as shown in Table 1. First, a schedulability analysis shows us that this task set is unschedulable in one processor as $U = \Sigma(C_i/P_i) = 1.60$. Our aim is to see whether this set can be scheduled on two processors using our approach.

Table 1. An Example Task Set

T_i	C_i	D_i	P_i
A	16	30	30
B	24	40	40
C	33	60	60

Assume the threads for each task are as shown in Fig 3 with unity communication costs. The graph partitioning algorithm produces the partitions as shown in Table 2 with weights set equal. The initial centers are chosen randomly as 3 and 4 for Task A; 2 and 7 for Task B; and 3 and 6 for Task C. At each iteration, the combination heuristic $H = W_1 * EDF + W_2 * HEM$ is applied around these centers for the collapse operation. We then input these data which represents the rough schedule to the task thread sequencing algorithm (TSA) of Fig. 2 which produces the output depicted in Table 3. We can now calculate the worst case response times for each task component as shown and conclude that this task thread sets are schedulable as $R_{ij} \leq d_i$, $\forall i$. We have assumed that the blocking times for threads and the network delays are negligible.

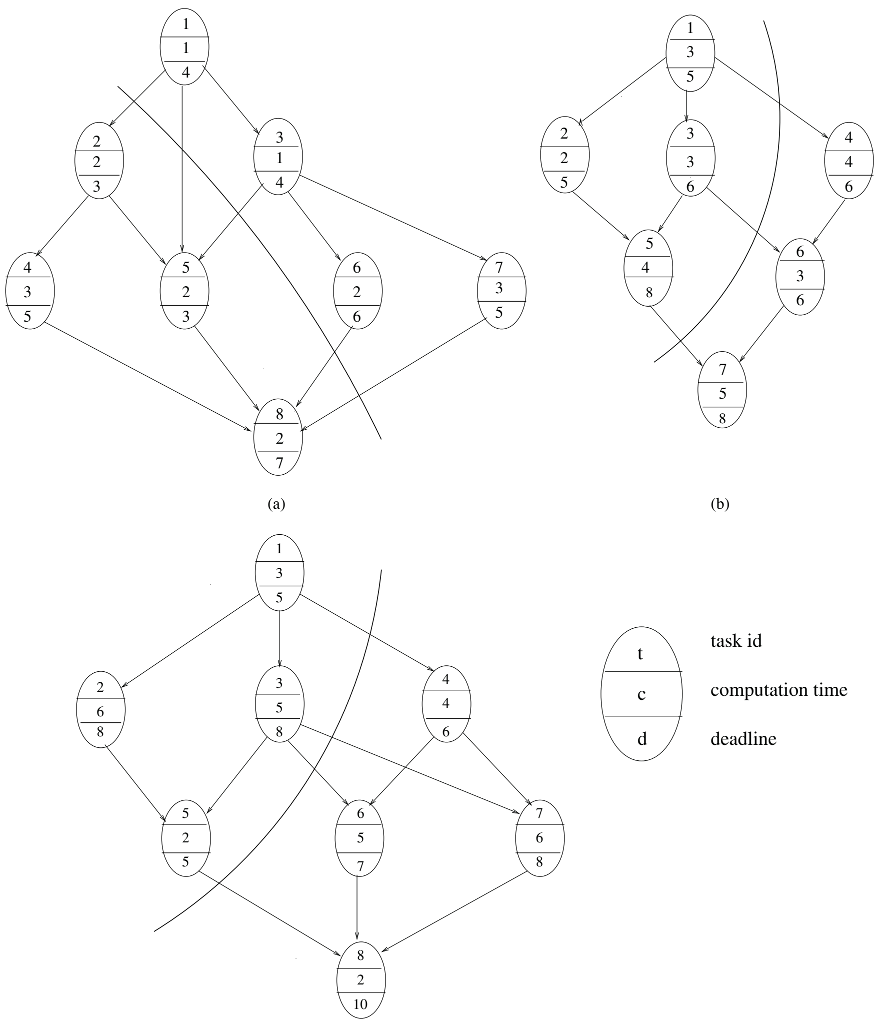

Fig. 3. Tasks: a) Task A b) Task B c) Task C

Table 2. TGPA Iterations

Tasks	*Processor*	*Initial*	$iter_1$	$iter_2$	$iter_3$
A	P1	3	3-1	3-1-7	3-1-7-6
	P2	4	4-2	4-2-5	4-2-5-8
B	P1	2	2-1	2-1-3	2-1-3-5
	P2	7	7-6	7-6-4	7-6-4
C	P1	3	3-1	3-1-2	3-1-2-5
	P2	6	6-4	6-4-7	6-4-7-8

Table 3. TSA Output

Tasks	*Processor*	*TSA Output*	Σc_{ij}	R_{ij}	P_i
A	P1	1≺3≺6≺7	7	7	30
	P2	2≺5≺4≺8	9	9	30
B	P1	1≺2≺3≺5	12	19	40
	P2	4≺6≺7	12	21	40
C	P1	1≺3≺2≺5	16	42	60
	P2	4≺7≺6≺8	16	46	60

The final schedule is given in Fig. 4. The allocation that meets the deadlines of every thread of every task will repeat every 120 units which is the *hyperperiod* (lowest common multiple of the three task periods).

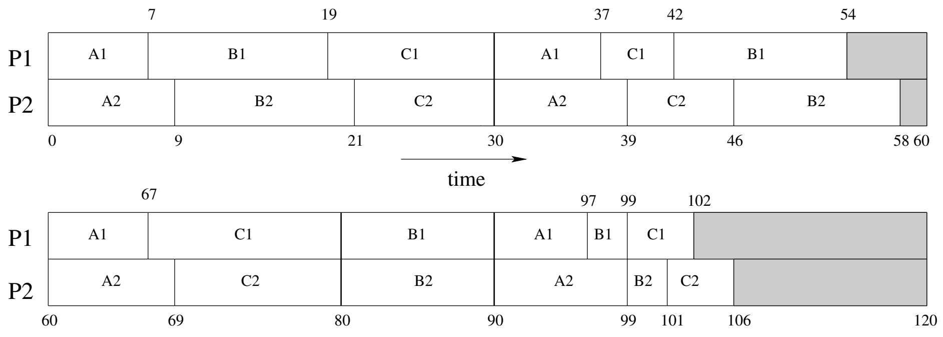

Fig. 4. The Allocation

6 Conclusions

We have presented algorithms based on graph partitioning to yield feasible schedules of real-time tasks to distributed processors. To our knowledge, there is not a significant research effort to accomplish this task using graph partitioning methods. For tasks allocated to different processors, the bounded message delays need to be considered for this model to yield effective results. Also, we need to determine relative weights of the EDF and HEM heuristics to give attainable schedules per processor while partitioning the task graph. Further experimental studies are neceessary to evaluate the proposed algorithms and also different scenarios including threads of different tasks accessing shared resources are needed. It is possible to perform TGPA in parallel for all tasks independently and also for each fixed center collapses within the threads.

References

1. Audsley, N.C., Burns, A., Richardson, M., Wellings, A.: Hard Real-time Scheduling: The Deadline Monotonic Approach, Procs. of IEEE Workshop on Real-Time Operating Systems and Software, (1991), 133-137
2. Dipippo, L. C. et al.: Scheduling and Priority Mapping for Static Real-Time Middleware, Real-time Systems, 20(2), (2001), 155-182
3. Erciyes, K, Marshall, G., A Cluster Based Hierarchical Routing Protocol for Mobile Networks, Springer Verlag, Lecture Notes in Computer Science, ICCSA(3), (2004), 528-537
4. Karypis G., J., Kumar J., Analysis of Multilevel Graph Partitioning, University of Minnesota, Dept. of Computer Science, Tech. Report 95-037
5. Liu J., Layland J.: Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment, Journal of the ACM, 20(1), (1973), 40-61
6. Lehoczky, J.P., Sha, L., Ding, Y.: The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour, Procs. of IEEE Real-Time Systems Symposium, (1989), 166-171
7. Lin, M., Yang, L.T.: Hybrid Genetic Algorithms for Scheduling Partially Ordered Tasks in a Multi-processor Environment, Procs. of 6th International Conference on Real-Time Computer Systems and Applications. (1999), 382-387
8. Lui, S., Rajkumar R., Lehoczky, J. P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, 39(9), (1990), 1175-1185
9. Lui S., Rajkumar R., Shrish S.: Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems, Procs. of the IEEE, 82(1), (1994), 68-82
10. Natale, M.D., Stankovic, J.A.: Scheduling Distributed Real-time Tasks with Minimum Jitter, IEEE Transactions on Computers 49(4), (2000), 303-316
11. Oh, J, Wu, C.: Genetic-algorithm-based real-time task scheduling with multiple goals, Journal of Systems and Software, 71(3), (2004), 245-258
12. Chu, C.: Parallel Machine Scheduling to Minimize Total Tardiness. International Journal of Production Economics, 76(3), (2002), 265-279
13. Xu. J., Parnas, D.: Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, IEEE Trans. On Software Engineering, 16(3), (1990), 360-369