Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-Time Systems Using Shared Buffers *

Sharath Kodase, Shige Wang, Zonghua Gu, Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
email:{skodase,wangsg,zgu,kgshin}@eecs.umich.edu

Abstract

Scheduling precedence-constrained tasks in a distributed real-time system is an NP-hard problem. As a result, the task allocation and scheduling algorithms that use these heuristics do not scale when applied to large distributed systems. In this paper, we propose a novel approach that eliminates inter-task dependencies using shared buffers between dependent tasks. The system correctness, with respect to data-dependency, is ensured by having each dependent task poll the shared buffers at a fixed rate. Tasks can, therefore, be allocated and scheduled independently of their predecessors. To meet the timing constraints of the original dependent-task system, we have developed a method to iteratively derive the polling rates based on end-to-end deadline constraints. The overheads associated with the shared buffers and the polling mechanism are minimized by clustering tasks according to their communication and timing constraints. Our simulation results with the task allocation based on a simple first-fit bin packing algorithm showed that the proposed approach scales almost linearly with the system size, and clustering tasks greatly reduces the polling overhead.

1 Introduction

Distributed real-time systems are becoming larger and more complex as the scope of real-time computing extends to more demanding and challenging applications. To meet timing and schedulability constraints in such applications, task allocation and schedule (TAS) decisions, must be made at design time. The problem of allocating and scheduling precedence-constrained tasks on processors in a distributed real-time system is NP-hard. For such tasks, even the problem of determining if a given allocation of tasks to processors satisfies the timing constraints is NP-hard. As such, heuristics for determining the timing satisfiability of a task allocation can be very complex [12], or involve generating the entire schedule in one planning cycle [1][8, 9]. These approaches are computationally-intensive and can take very long time to find a solution.

Existing heuristics for task allocation and scheduling (TAS) are based on artificial intelligence search techniques like branch-and-bound [8], simulated annealing [2, 7, 11] and tabu search [6]. These algorithms obtain approximate or near-optimal solutions to the TAS problem by iteratively evaluating different points in the search space. The number of points to evaluate in the search space is large even when dealing with moderately-sized task sets. Therefore, these algorithms do not scale well when applied to large distributed real-time systems. The task system generated with such an approach suffers from difficulties in checking if the timing and schedulability constraints are met during software integration and testing phases. This results in higher development costs, a longer development cycle, and sometimes even redesign.

In this paper, we propose a new approach that eliminates the inter-task dependencies existing in a large-scale applications by introducing shared buffers between pairs of dependent tasks. Successor tasks are decoupled from their predecessors as they need only access to the buffers. The correctness, with respect to data-dependency, of the system is preserved by having successor tasks poll the shared buffers at predefined rates. Thus, the precedence-constrained task system is transformed into a system consisting solely of independent tasks. This transformed system of independent tasks is much easier for their allocation and scheduling than

*The work reported in this paper was supported in part by DARPA under the US AFRL Contract No. F33615-00-C-1706.

[1]The planning cycle is equal to the LCM (least common multiple) of periods of all tasks on a processor.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

the original system.

The rest of this paper is organized as follows. Section 2 presents the system model and states the problem of resolving dependencies with shared buffers. Section 3 details the shared-buffer approach including the methods of deriving the polling rates for TAS and reducing the resultant overheads. Section 4 evaluates the proposed approach and presents an example application. The paper concludes with Section 5.

2 System Model

A real-time task system A is defined as a set T of intercommunicating (hence inter-dependent) real-time tasks. The system can be represented as a directed acyclic graph (DAG) G_A, where a node represents a task and a link represents the dependency between two tasks. Given any two tasks T_i and T_j with a directed link from T_i to T_j, we say T_j is immediately dependent on T_i. T_i is called a *predecessor* of T_j, and T_j is called a *successor* of T_i. Such a predecessor–successor relationship is transitive.

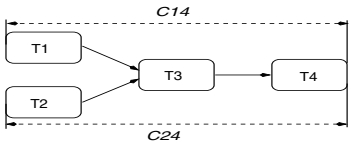
In G_A, nodes without predecessors are called *input tasks*. Similarly, nodes without successors are called *output tasks*. Typically, input tasks are triggered periodically or sporadically. We make no distinction between these two types of invocations, and denote the interval between two successive invocations of a task T_i by p_{T_i}. For any output task T_o, there should exist at least one path P_{io} from an input task T_i to it in G_A. Such a path between a pair of input and output tasks are subject to a timing constraint, denoted by d_{io}, which specifies that T_o should be finished within d_{io} units of time after T_i is released.

Given the system model defined as above, our goal is to transform the system A with dependent task set T to an equivalent system A' with task set T', where (a) the task set $|T'| = |T|$; (b) each task T_i in T has a corresponding task T_i' in T'; (c) tasks in T' are all independent; and (d) if there exists a directed link L_{ij}: $T_i \rightarrow T_j$ in A, there is a buffer B_{ij} which T_i' writes its outputs to, and T_j' reads its input from. T_j' periodically checks B_{ij} for updates from T_i'. The period at which T_j' checks for buffer updates is called the *polling period* of T_j'. The transformation process ensures the function of the original system A is preserved in A', and also eliminates dependencies by introducing the shared buffer B_{ij}. The resulting system A' can easily fit the assumptions of most TAS algorithms, and thus, any of them can be used in our system design and analysis.

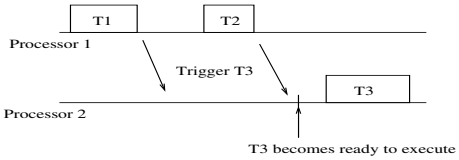
The transformation from A to A' should be done in such a way that the schedulability and timing constraints of the original system A are preserved in the transformed system A'. In other words, if A' is schedulable, then so is A. Our approach preserves the timing constraints and schedulability by deriving the polling periods for tasks in A' based on A's timing constraints and schedulability considerations.

3 System Transformation Using Shared Buffers

Figures 1 and 2 give an example system with dependent tasks and the corresponding transformed system using shared buffers. The task graph of the original system contains 4 tasks, where T_3 depends on the outputs of both T_1 and T_2, and T_4 depends on the output from T_3, as shown in Figure 1(a). The system is subject to two timing constraints C_{14} and C_{24}. The runtime scenario of the original system is shown in Figure 1(b).



(a) Original task graph with 4 tasks.



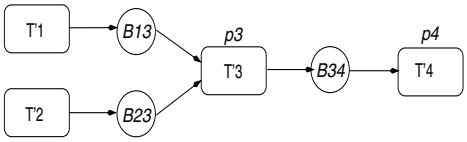
(b) Scheduling with dependencies on 2 processors.

Figure 1. An example system with dependent tasks.

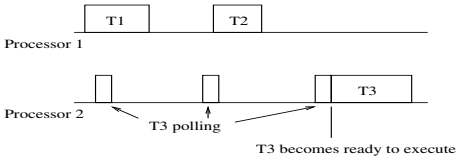
The system is transformed into a system with independent tasks as shown in Figure 2(a). The transformed system contains the original 4 tasks and 3 shared buffers. The polling rates p_3 and p_4 are assigned to T_3' and T_4' such that constraints C_{14} and C_{24} are satisfied. Tasks are triggered independently, as shown in Figure 2(b).

In the transformed system, the explicit triggering mechanism used to meet the precedence constraints in the original system is replaced by an implicit trigger stored and updated as flags along with the shared buffers. The successor tasks decide when to process their input buffers based on these flags. Given the original system computed or output a value *var* with time period, P_{var}, the transformation only changes the internal mechanism that determines how the sequences of tasks determining *var* are triggered. The rest of the system behavior like the input rate of data into the system, or the output rate of data to the environment remains

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE



(a) Task graph with independent tasks after transformation.



(b) Schedule of transformed tasks on 2 processors.

Figure 2. The system after transformation using shared buffers.

unchanged.

3.1 Derivation of shared-buffer polling rates

In the proposed approach, tasks will check the shared buffers for updates at some fixed intervals after the system transformation. To ensure that non-functional constraints such as task deadlines, rates and schedulability are met after the transformation, the polling rates for tasks in the system A' have to be determined from the timing constraints C_A in the original system A. For any successor task in the transformed system, the worst-case update-detection delay occurs when the shared buffer is updated immediately after the task checks it. Since the task can only detect the updates in the next polling cycle in such cases, the delay can be at most one polling period. Therefore, the condition under which every task in transformed system satisfies C_A can be expressed as:

$$poll_{T_i'} + r_{T_i'} \leq d_{T_i'} - s_{T_i'} \tag{1}$$

where $poll_{T_i'}$ is the task's polling period, $r_{T_i'}$ is the response time (also called the *completion time*), $d_{T_i'}$ is the deadline for execution of T_i', and $s_{T_i'}$ is the start time of T_i'. The term $d_{T_i'} - s_{T_i'}$ defines the maximum allowable processing time for T_i'. $s_{T_i'}$ can be derived from the deadlines of T_i''s predecessors as:

$$s_{T_i'} = \max\{d_{T_k'} : T_k' \text{ is an immediate predecessor of } T_i'\}.$$

The deadlines for each task in A can be derived from C_A using a modified version of the deadline-distribution algorithms in [5].

To derive the polling period of T_i' using Eq. (1), we need to know the response time of T_i'. Such response time depends on the allocation of T_i', which, in turn, depends on the polling period for each task. This circular dependency between the response time and the polling period for each task can be resolved by iteratively determining solutions for both. We can set the initial polling period of T_i' to p_{T_i} and perform TAS. After obtaining $r_{T_i'}$ from TAS, Eq. (1) can be checked. For those tasks that do not pass the check, their polling periods should be revised systematically and TAS should be repeated. This process of assigning polling periods to tasks and performing TAS should be iterated until all tasks in T_i' satisfy Eq. (1).

If no task in T' satisfies Eq. (1), it indicates that the worst-case update-detection delay for the task is longer than required. To reduce the detection delay, the task polling period should be reduced. We achieve using Eq. (2):

$$\begin{aligned} excess_{T_i'} &= p_{T_i'} + r_{T_i'} - d_{T_i'} + s_{T_i'} \\ poll'_{T_i'} &= poll_{T_i'} - excess_{T_i'}/2. \end{aligned} \tag{2}$$

where $poll'_{T_i'}$ is the new polling period of T_i', and *excess* is defined as the amount of time by which T_i' exceeds its maximum allowable execution time. Figure 3 provides the algorithm for polling period derivation. Since shortening the polling periods increases the system workload in each iteration and will eventually lead to an unschedulable task set, the algorithm is guaranteed to terminate in a finite number of iterations.

After transformation, schedulability analysis is required to ensure the system-wide timing correctness. To perform schedulability analysis, we replace a polling task by two tasks, one dedicated to the polling behavior, and the other to data processing, because of different execution times for polling operation and data processing. The one for data processing is only executed when the shared buffer is updated, and is released with a fixed offset equal to the polling period to ensure the data availability. These two tasks can then be considered as independent tasks and schedulability analysis can be performed using traditional techniques like fixed priority pre-emptive scheduling [1].

3.2 Polling overhead reduction

Resolving inter-task dependencies with shared buffers induces overheads to the final system as a result of tasks polling shared buffers. Such overheads include additional context switches, and blocking times for shared-buffer access and management (e.g., setting and updating flags). These overheads depend on the number of tasks in the system and their polling rates. Larger number of tasks and

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

```
Algorithm PP_D: Poll_Period_Derivation
Input: T': tasks in transformed system A',
    T: tasks in original system A,
    P: processors,
    C_A: original end-to-end timing constraints,
    poll_thresh: user specified polling period threshold for tasks in T'.
Output: {poll_T'_i}: polling periods for each task in T',
    Alloc: allocation of tasks in T' to processors in P.
Begin:
distribute timing constraints C_A over each task in T in A.
for each task T'_i in T', poll_T'_i = p_Ti.
Loop:
    allocate and schedule tasks T' on processors P → Alloc.
    if(every task T'_i in T' can be successfully scheduled)
    {
        if(every task T'_i satisfies Eq. (1))
            return successful, {poll_T'_i}, Alloc.
        else
        {
            for every task T'_j that fails Eq. (1)
            {
                decrease poll_T'_j using Eq. (2).
                if(poll_T'_j ≤ poll_thresh)
                    return failure.
            }
            goto Loop.
        } /* end-else */
    } /* end-if */
    else return failure.
end
```

Figure 3. Algorithm of polling period derivation.

higher polling rates introduce more context switches in a given period of time. The overhead of shared-buffer management also increases with the number of tasks and their polling rates because of frequent checking and updating of flags on shared buffers. In this work, we adopt task clustering to reduce the number of tasks and their polling overheads.

Task clustering partitions a set of tasks into several disjoint groups and uses explicit ordering to resolve inter-task dependencies within each group. We use pairwise clustering [9] and period-based clustering [10] for this, although other methods can be used as well. The number of tasks is further reduced using *local clustering*, in which tasks with precedent dependencies are clustered together if they are allocated to the same processor. Since the number of independently polling tasks is reduced, the system overhead is lowered due to fewer context switches and lower polling frequencies.

The overhead is further reduced by distributing deadlines over the task clusters. Each cluster is treated as a composite task in the deadline distribution algorithm. Since the system now contains smaller number of composite tasks, more slack can be allocated to each composite task. This will result in a larger value of $d_{T'_i} - s_{T'_i}$ in Eq. (1), which will consequently decrease the polling rates.

The polling overhead can be further reduced by reducing or eliminating blocking time using multiple-buffers [4] between predecessor and successor tasks.

4 Evaluation

We evaluated the effectiveness and efficiency of our approach using both simulation systems as well as a representative application.

Experiment configuration:

Since the goal of this work is to apply simple, polynomial time TAS algorithm, a simple TAS algorithm must be chosen first in the evaluation. We chose the first-fit algorithm [3] for task allocation. This algorithm orders tasks according to decreasing resource consumption (or task utilization) and allocates them, one by one, to processors. The priority assignment was based on deadline-monotonic policy with a higher priority assigned to a task with tighter deadline, and the holistic scheduling approach [1] was used for the schedulability analysis. [2]

Our algorithm for polling rate derivation partitions end-to-end timing constraints over individual tasks or composite tasks by using two well-known deadline-distribution heuristics: *pure slicing* and *normal slicing approach* [5]. The pure slicing approach distributes slack in the deadline according to the number of tasks in the directed acyclic graph, while the normal slicing approach distributes the deadline to component tasks in proportion to their execution times. Only *local clustering* is used in comparisons to evaluate the overhead reduction methods.

The simulated system we exercised contains a set of tasks whose number ranges randomly from 60 to 300, partitioned into $10 \sim 50$ groups, with each group consisting of $4 \sim 8$ dependent tasks.

Table 1 summarizes the selected algorithms and system parameters in the simulation.

In the simulation, we first constructed several task graphs with properties assigned according to Table 1. Then, they were transformed into independent tasks using the shared-buffer approach. Algorithm PP_D was executed to iteratively generate task allocation and scheduling, and provided the number of processors used to schedule the given task set, processor utilizations, polling overheads, and the number of iterations to generate a schedulable task set. This process was repeated 3 times with different random seeds and an average was taken on the obtained values.

Experimental results:

[2] Since we are only interested in the relative performance of the shared-buffer approach, the choices of algorithms will not affect the comparison results.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

To find a schedulable task allocation, the total number of schedulability checks performed in each iteration by the first-fit TAS is of order $\Theta(n * p)$, where n and p are the number of tasks and the number of processors, respectively. The total number of checks performed during the execution of Algorithm PP_D depends on the number of iterations required to converge to a solution. The number of iterations required to allocate tasks on processors with a different number of tasks in the task set is given in Figure 4(a). In this experiment, we fixed the slacks in all dependent-task graphs to a constant of 7 times of the execution time for all tasks. The results in Figure 4(a) show that the number of iterations increases almost linearly with the number of tasks. Irrespective of the number of tasks in the system, the number of iterations to generate a solution is bounded around 20, even in a graph of 250 tasks. This indicates that the TAS problem for a large task set can be solved in a short time using our approach.

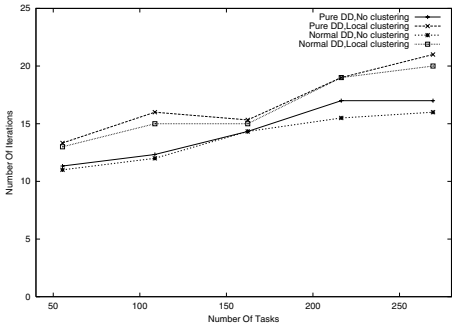
In each iteration, any task that does not satisfy Eq. (1) has its polling period reduced using Eq. (2). Since the polling period of a task can only decrease with each iteration until the task can be either satisfactorily scheduled, or below a specified threshold when the algorithm gives up, the number of iterations in which each task has its polling period revised, is bounded. This sets an upper limit to the number of iterations that the algorithm can perform, and this limit is linear in the number of tasks for a given slack.

The simulation results on the efficiency of using shared buffers are plotted in Figure 4(b), showing the number of processors required to make the task set schedulable using the first-fit algorithm. We compare the number of processors returned by Algorithm PP_D with that returned by the first-fit TAS on the same system, but without inter-task dependencies. We set the maximum number of tasks in this experiment to 220 tasks.

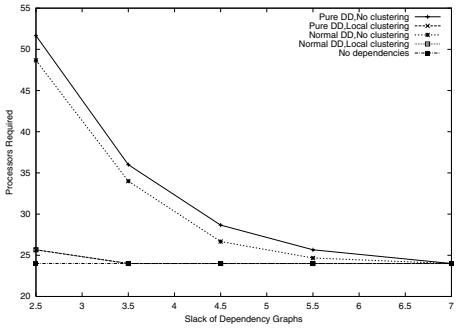
The results in Figure 4(b) show that without clustering, allocation algorithms using the polling method perform worse when tasks have low slacks in their deadlines. This is because the polling rate for tasks gets higher after transformation when the slacks are low, thus wasting more processor time for polling. But for larger system slacks, the processors returned by the polling approach asymptotically meets that of scheduling the system without any inter-task dependencies. The polling rates for dependent tasks decrease as the system slacks increase. In such cases, truly independent tasks require fewer processors for them to be schedulable. This implies that our approach be more useful when tasks have more slack and longer deadlines, which is commonly the case in large-scale distributed systems.

Figure 4(b) also shows the efficacy of task clustering in terms of polling overhead reduction. Even at low system slacks, if the local clustering is used, the number of processors required to schedule the task system is only marginally higher than that required to schedule the same system without task dependencies. This indicates that task clustering is indispensable to the shared-buffer approach.

Figure 5(a) shows the polling overheads before and after task clustering. The polling overhead is defined as:



(a) Number of iterations vs. number of tasks



(b) Number of processors required vs. slack

Figure 4. Simulation results for different numbers of iterations and processors required.

Deadline-distribution	Pure slicing, Normal slicing
Task clustering	Local clustering
Task allocation	First-fit bin packing algorithm
Scheduling	Static priority pre-emptive scheduling
Average out degree of each task	2
Average in degree of each task	2
Execution time of each task	$5 \sim 10$ ms
Period of a dependency graph	$50 \sim 150$ ms
Slack for a dependency graph	$2.5 \sim 7$ times the sum of execution times of tasks in the graph
Polling overhead	$5 \sim 10\%$ of single task execution times

Table 1. Algorithm selection and graph characteristics in the simulation system.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

$$\text{polling overhead in } \% = \frac{\frac{\sum_{p \in P} \sum_{T'_j \in TS_p} \frac{pollexec_{T'_j}}{poll_{T'_j}}}{|P|}}{\frac{\sum_{p \in P} \sum_{T'_j \in TS_p} \{\frac{pollexec_{T'_j}}{poll_{T'_j}} + \frac{e_{T'_j}}{p_{T'_j}}\}}{|P|}}$$

where P is the set of processors, TS_p the set of tasks allocated on p, $pollexec_{T'_i}$ the execution time for polling T'_i, $poll_{T'_i}$ the polling period of T'_i, $p_{T'_i}$ the invocation period of T'_i, and $e_{T'_i}$ the execution time for data processing of T'_i. The numerator in the above equation denotes the polling utilization averaged over all tasks and processors. The denominator in the equation denotes the sum of polling and task utilizations averaged over all tasks and processors.

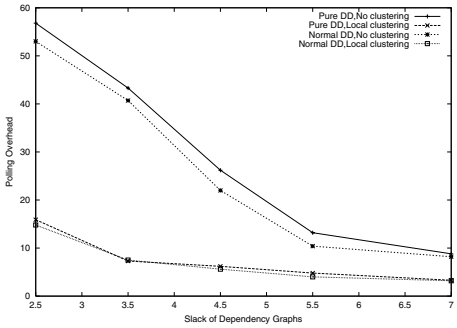


Figure 5. Simulation results for polling overheads.

From Figure 5, we observe that the overhead without task clustering is $2 \sim 3$ times higher than that with local clustering. We also observed that the polling overheads decrease as the system slacks increase. This is also caused by lowering the polling rates of tasks with increased slack. With task clustering, the polling overheads are reduced and thus more acceptable as they are around less than 15%, even at low system slacks.

We also observed that the normal deadline-distribution algorithm tends to introduce smaller polling overheads, use fewer processors, and assign lower polling rates for dependent tasks than the pure deadline distribution when task clustering is not used. When task clustering was used, both the normal and pure deadline-distribution algorithms tend to generate similar results on these metrics.

Comparison with regular TAS:

As the TAS for precedence-constrained periodic tasks in a distributed system is NP-complete, existing heuristics for determining the satisfiability of a task allocation can be very complex [12], or involve generating the entire schedule up to the planning cycle — LCM (least common multiple) of periods — of all tasks on each processor [8, 9]. These approaches are computationally very intensive and can take very long time. Since such timing satisfiability checks are invoked at each step of any TAS algorithm, the running times can be very large even for moderately-sized task sets.

In the polling method, the global end-to-end timing constraints are broken to individual timing constraints on component tasks. Each task is made independent of others using shared buffers. Consequently, determining the satisfiability of a given deadline allocation to tasks only involves schedulability check on each processor. Since such checks for independent periodic tasks can be done accurately and very fast, TAS using the polling method is much faster than the usual non-polling approach.

To evaluate this, we compared the running times for TAS using the polling method (polling TAS) with TAS not using polling method, but allocating and scheduling based on dependencies (regular TAS). For polling method, we used normal deadline distribution algorithm. For regular TAS, we used a simple depth-first search (DFS) algorithm that terminates after finding the first solution that satisfies the timing constraints. To determine timing satisfiability, we used a simple algorithm that determines the response time of each task using holistic analysis [1], and then determines the end-to-end response times using the response times of individual component tasks. To see how fast this simple check is, we determined the time taken for the satisfiability tests for an example task set in [12], on a sun workstation. Our simple check took 0.0019 second whereas the iterative method in [12] took 0.13 second. Using this simple check for DFS TAS, we measured the times taken to allocate and schedule 15 task graphs with characteristics as given in Table 1, on a 1.2GHz AMD Athlon processor with 256MB RAM. The time taken for both approaches for systems with different slacks is shown in Table 2.

Approach	System slack=5.0	System slack=7.0
Polling method	1.5 sec	1.3 sec
DFS method	2334 sec	2111 sec

Table 2. Polling method and regular TAS

Even with a simple satisfiability check, the DFS algorithm takes a much longer time than the polling method. We also found that since DFS is not an optimal algorithm, it takes more processors than the polling method with first-fit bin packing allocation to schedule the tasks.

A more rigorous timing check like in [12], and to optimize TAS for a metric like load balancing, or to minimize processors, would increase the time for TAS significantly when dependencies are considered. With the polling

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

method, however, it is easier and much faster to perform TAS to optimize for metrics like load balancing, or minimizing processors owing to its simpler timing satisfiability checks and task independence.

However, because of the overheads associated with the polling method, its solutions may be inferior to optimum solutions obtained directly. To evaluate how the polling method compares with an (optimum) regular TAS, we compared the number of processors required to schedule 9 task graphs using the polling method and an exhaustive search TAS algorithm. The characteristics of the task graphs are given in Table 1. Table 3 shows the number of processors required and their average utilizations.

Approach	System slack=2.0	System slack=5.0
Polling method	6.67 processors, 0.82	6 processors, 0.80
Optimum TAS	6 processors, 0.76	6 processors, 0.76

Table 3. Polling method and optimum TAS algorithm

For a system slack of 2.0, due to its overheads, the polling method requires, on average, an extra processor to schedule the task set. However, when the system slack is higher at 5.0, the polling method requires the same number of processors as the regular TAS method. As described and evaluated above, a higher slack in deadlines decreases the polling overhead, making the performance of the TAS algorithm using polling closer to the optimum.

4.1 Automotive Control Applications

We further evaluated the usefulness of our proposed approach in real applications. We chose an automotive vehicle control system as a representative application to apply the shared-buffer approach. The task graph of the vehicle control system is given in Figure 6. Although this system contains only a small number of tasks and their dependencies, the timing constraints are stringent and dependencies are representative.

The system contains two controllers managing engine operations and distance between vehicles. The two controllers are to be scheduled on two Motorola 555 processors (MPC 1 and MPC 2), with both running OSEKWorks operating system. The processors are connected together by a high-speed controller area network (CAN) of 1 Mbps. Sensors for engine speed, vehicle speed, crank angle, brake position, radar, and steering wheel position are associated with MPC 1, and actuators for throttle, transmission, brake and steering wheel actuators are bounded to MPC 2. The sensor tasks, TS_CKP, TS_ENG, TS_E_T, TS_TRAN, VEH_LAT and RUNMODULES need to run on MPC 1, where actuator tasks TA_ENG, TA_TRAN, CAN_BRAKE and HST have

Tasks	Exec time (ms)	Bound Processor
TS_CKP	0.10	MPC 1
TS_ENG	0.13	MPC 1
TS_E_T	0.20	MPC 1
TS_TRAN	0.25	MPC 1
TC_CKP	0.13	
TC_ENG	0.08	
TC_TRAN	0.05	
TP_ENG	0.07	
TP_TRAN	0.14	
TA_ENG	0.13	MPC 2
TA_TRAN	0.09	MPC 2
VEH_LAT	0.074	MPC 1
BUTTONS	0.08	
HMI	0.07	MPC 2
HST	0.09	MPC 2
CAN_READ	0.104	MPC 1
VEH_IOLS	0.345	
ENG_SPDLS	0.760	
CAN_BRAKE	0.08	MPC 2
RUNMODULES	0.101	MPC 1
DPC_SEND	0.08	

Table 4. Timing characteristics of the vehicle control tasks.

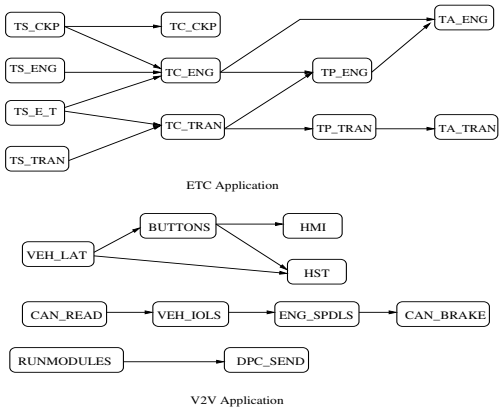


Figure 6. Automotive control applications.

to be scheduled on MPC 2. The real-time characteristics of tasks are as shown in Table 4.

The period at which fresh data enters tasks TS_CKP, TS_ENG, TS_E_T, TS_TRAN is 5ms. The end-to-end deadline for engine control tasks to process inputs and generate outputs should be within 5ms. Steering wheel position data is processed by VEH_LAT at a period of 4ms. Deadline for processing steering wheel data is 4ms. CAN_READ and RUNMODULES tasks are required to process their inputs at periods of 10ms and 4ms, respectively. The end-to-end deadlines for processing these data are the same as periods.

For this scenario, we applied our iterative TAS using the first-fit bin packing algorithm with fixed-priority scheduling and deadline-monotonic priority assignment. The polling overhead including all OS overheads (e.g., context switch and timer overhead) is measured to be 0.01 ms for both processors. The analysis results for the normal deadline-

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

Task	Polling Period (ms)	Response Time (ms)	Processor
TS_CKP	0.45	0.13	MPC 1
TS_ENG	0.45	0.26	MPC 1
TS_E_T	0.45	0.46	MPC 1
TS_TRAN	0.45	0.72	MPC 1
TC_CKP	0.85	0.37	MPC 2
TC_ENG	0.85	0.16	MPC 2
TC_TRAN	0.85	0.08	MPC 2
TP_ENG	0.85	0.44	MPC 2
TP_TRAN	0.85	0.58	MPC 2
TA_ENG	0.85	1.40	MPC 2
TA_TRAN	0.85	1.26	MPC 2
VEH_LAT	0.56	0.79	MPC 1
BUTTONS	1.11	0.24	MPC 2
HMI	1.11	1.08	MPC 2
HST	1.11	1.17	MPC 2
CAN_READ	1.27	1.01	MPC 1
VEH_IOLS	1.52	1.01	MPC 2
ENG_SPDLS	1.52	2.18	MPC 2
CAN_BRAKE	1.52	2.27	MPC 2
RUNMODULES	1.11	0.91	MPC 1
DPC_SEND	1.34	0.66	MPC 2

Table 5. The results using the normal deadline-distribution algorithm.

distribution algorithm are shown in Table 5.

Our algorithm took 11 iterations to complete. The polling periods for tasks on the same processor are assigned the same because of task clustering. The polling overhead for normal deadline distribution turns out to be 4.6%.

The data transmitted between these tasks is very small (in the order of at most 4–8 bytes). For this reason, we ignore the inter-processor communication. However, this can be accounted for by considering the network as a processor and any inter-processor communication as a task to be scheduled on the network. The inter-processor communication tasks will depend on task allocation and will be different for each allocation. Determining of these tasks and scheduling them on the network processor can be handled during TAS algorithm itself. However, Algorithm PP_D will remain same and equally applicable.

5 Conclusions

In this paper, we proposed a new approach to eliminating inter-task dependencies by introducing shared buffers between dependent tasks, and assigning a proper polling rate for each dependent task to ensure the end-to-end timing constraints are met. By making dependent tasks independent, existing efficient and fast TAS algorithms can be used to find a solution in polynomial time for large distributed systems. We developed an algorithm that iteratively assigns the polling rates for tasks during TAS.

The results based on both simulation and real application of an automotive vehicle control system have shown that significant scalability benefits for TAS algorithms can be gained using the proposed approach, and the polling overhead was greatly reduced by task clustering. As the number of tasks in the system increases, the number of iterations of our algorithm and the number of schedulability checks increase only linearly. The simulation results have also indicated that our approach is more effective when there is more slack in the deadlines of dependent tasks.

References

[1] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[2] J. Beck and D. Siewiorek. Simulated annealing applied to multicomputer task allocation and processor specification. *8th IEEE Symposium on Parallel and Distributed Processing*, pages 232–239, 1996.

[3] R. Graham. *Bounds on the performance of scheduling algorithms*, pages 165–227. John Wiley and Sons, 1976.

[4] H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. *Usenix Annual Technical Conference*, June 2002.

[5] J. Jonsson and K. Shin. Deadline assignment in distributed hard real-time systems with relaxed locality constraints. In *17th International Conference on Distributed Computing Systems*, pages 27–30, Baltimore, MD, May 1997. IEEE Comput. Society.

[6] M. Lin, L. Karlsson, and L. Yang. Heuristic techniques: Scheduling partially ordered tasks in a multi-processor enviroment with tabu search and genetic algorithms. *7th International Conference on Parallel and Distributed Systems*, pages 515–523, 2000.

[7] M. D. Natale and J. Stankovic. A simulated annealing for multiprocessor scheduling. Technical Report 10, ARTS lab, S. Anna, Pisa, 1995.

[8] T. Peng, K. Shin, and T. Abdelzaher. Assignment and scheduling of communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), 1997.

[9] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, April 1995.

[10] A. Tarek and K. Shin. Period based load partitioning and assignment for large real time applications. *IEEE Transactions on Computers*, 49(1):81–87, January 2000.

[11] B. Wells and C. Carroll. An augmented approach to task allocation: combining simulated annealing with list-based hueristics. *Euromicro workshop on Parallel and Distributed Processing*, pages 508–515, 1993.

[12] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, November 1998.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE