Dynamic Scheduling for Mobile Robots

Tucker Balch, Harold Forbes and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA

tucker@cc.gatech.edu
fax:(404)853-0957

***Abstract*—This research concerns efficient multiprocessor threads-based implementation of reactive navigation for mobile robots. We present two important results: 1) Performance is improved significantly when CPU time allocated to individual navigational threads is adjusted dynamically according to a heuristic measure of their importance. 2) To implement this strategy, we present a multiprocessor scheduler design which can dynamically schedule navigational threads. The experiments were conducted in simulation on a BBN Butterfly and a KSR1 (shared memory multiprocessors). Speedups found for this example should extend to more complex navigational strategies as long as a heuristic measure of thread importance is available.**

I. Threaded Reactive Robot Navigation

The specific task examined in our research is robot navigation to a known goal position across an unmapped world potentially cluttered with obstacles. Many robot control systems have been proposed as solutions to this problem (e.g. [1, 6, 4, 7]). The system implemented in this research is based on the Autonomous Robot Architecture (AuRA) [2].

AuRA consists of both reactive and deliberative components. The deliberative component sets high level goals and selects appropriate behaviors to achieve them. The reactive component of AuRA executes the selected behaviors which are typified by tight sensor to actuator coupling. This research concerns the reactive component only.

Motor schemas are the basic unit of behavioral control in AuRA. Several schemas may be active as the robot navigates. Such schemas are independent processes that combine to generate an overall navigational behavior. Motor schemas take input from specialized perceptual schemas that process sensor data. Each motor schema generates a movement vector. These output vectors are summed, then normalized. The result is transmitted to the robot (or simulated robot) for execution.

Several systems utilizing this approach have been instantiated at Georgia Tech in simulation [?, 5], on mobile robots [?] and for multiagent research [?]. In [?], Collins describes a multiprocessor implementation of schema-based reactive system. His implementation, however, did not address the parallel execution of schemas. This implementation of AuRA is the first in which schemas run concurrently on multiple processors.

The system is implemented on a BBN Butterfly using the Cthreads library for parallel programming [9]. The various motor and perceptual schemas are instantiated as individual threads which communicate using shared memory. At execution time the following threads are activated:

- **Avoid-static-obstacle**: One instance of this motor schema is generated for each obstacle. All instances are independent and concurrently executable.
- **Move-to-goal**: A motor schema.
- **Noise**: A motor schema.
- **Move-robot**: References the outputs of the motor schemas and effects robot movement.
- **Monitor**: Terminates processing when the robot reaches the goal.

Other than mutual exclusion locks on shared data, there is no explicit thread synchronization. The following information is shared between threads:

- **Robot Position**: In cartesian coordinates.
- **Force On Robot**: Each motor schema adds its own component to this force. The result is added to the robot's position by the **Move-robot** thread.
- **Completion Flag**: Set to true by **Monitor** when the mission is complete.

Figure 1 presents the sample navigational task used in our experimentation. The robot is to navigate from the start in the lower left, to the goal in the upper right. Obstacles are represented by black circles. The resultant path is shown by the black line.

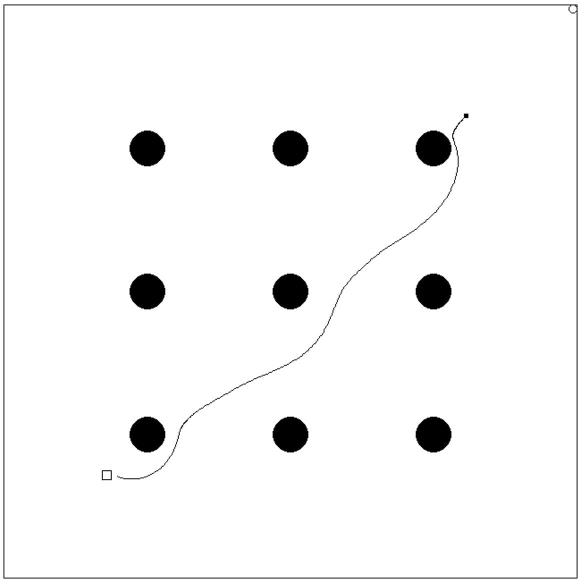

Fig. 1. Example Navigational Problem

II. Experimental Evaluation

The system is first tested using best effort scheduling, where all threads representing schemas are run in a round-robin fashion. Experimental runs are conducted with 1 to 13 processors. All runs assume the external environment shown in Figure 1, which includes 9 obstacles between the robot's starting point and its goal. Two performance metrics are recorded for each run: path length and execution time.

Since obstacle locations are not revealed a priori an optimal path cannot be precomputed. In fact, the robot is only allowed to use current sensor inputs for movement selection (this is the spirit of purely reactive control). Path length then reflects the distance the robot travelled to the goal. Execution time is the time it took for the robot to traverse this distance. The two metrics are distinct. More optimal paths may demand computational resources that would otherwise be used for obstacle avoidance. Since these resources are not available, the robot must slow down so as not to run into anything.

Figure 2 depicts the run times for reactive navigation when using multiple processors for schema execution. The speedup gains drop off as the number of processors approaches nine. This is due to contention for shared memory and synchronization overhead. Figure 3 depicts the resultant distance the robot followed to the goal as the number of processors is increased. The importance of this graph is that it shows path length does not degrade with more processors. We conclude that parallelism is an effective means for improvement of the execution speed of a schema-based navigation system.

While the experimental results of Figure 2 are encouraging one interesting insight is that the frequent execution of schemas that do not immediately affect the robot's current operation is both unnecessary and degrades performance by causing contention. Therefore, obstacle schemas should be scheduled dynamically so that their execution is delayed according to their effect on the robot's path. Figure 2 shows that from 18% (in the uniprocessor case) to 50% percent savings may be achieved when dynamic scheduling is used. These results are achieved by scheduling the **Avoid-static-obstacle** threads at the earliest time at which the associated obstacle could significantly affect the robot's path.

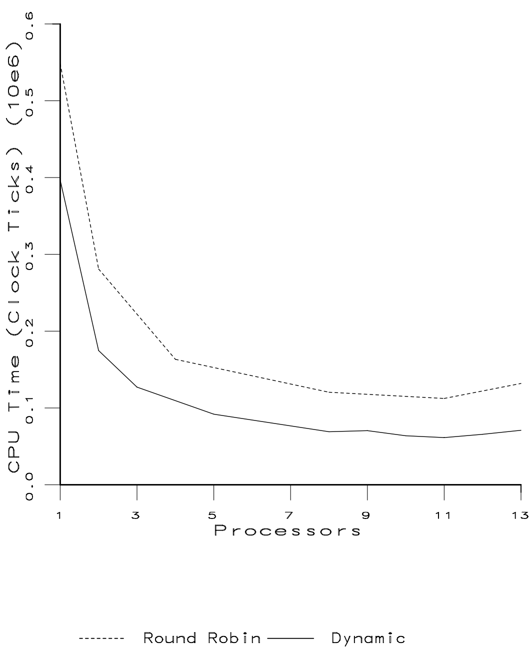

Fig. 2. Run times for round robin and dynamic scheduling of navigational threads on 1 to 13 processors

III. Guaranteed Dynamic Scheduling

The object of our research regarding dynamic schema scheduling is to ensure robot safety by guaranteeing the varying start times and hard deadlines required by the reactive navigation system. Such guarantees may be achieved by: 1. using strict round-robin scheduling with sufficiently fast cycle-time, 2: dynamically removing and appending an obstacle's thread in the run queue based on the obstacle danger and the worst case run queue cycle time, or 3: using explicit scheduling algorithms and schedule representation to guarantee future execution times. This research has shown that option 2 is better than option 1, and it may be that further performance improvements can be achieved by option 3. For a given safety

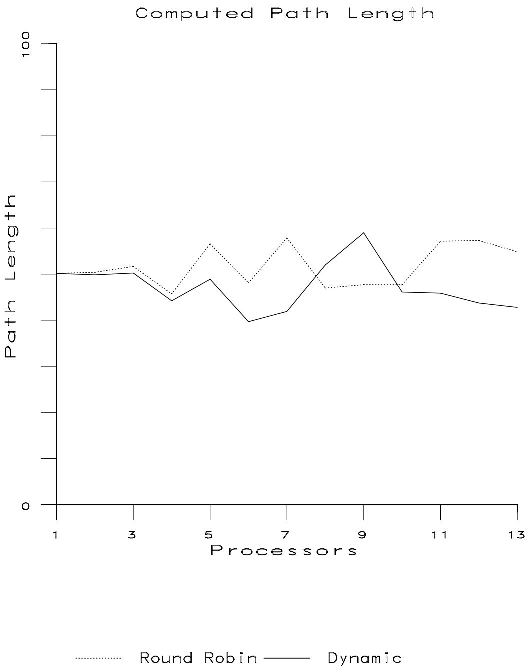

Fig. 3. Path length for runs with round robin and dynamic scheduling for navigational threads on 1 to 13 processors

level, guaranteed scheduling would allow the programmer to increase the period of an obstacle schema because he would not have to consider the worst case execution delay due to the size of the run queue.

The task timing model appropriate for guaranteed schema schedulability decisions is the triple:(earliest start time, maximum run time, deadline). The best effort scheduling described above calculates the the minimum amount of time until an obstacle can significantly affect the robot's path. This point in time defines the schema's deadline. Schema execution time can easily be calculated since it tends to be fairly data independent. In this application, the earliest start time is very flexible. It could any time before (deadline - run time). However, since the effect of a schema execution on the path of the robot is directly related to the distance between the robot and the obstacle, the starting time should only be early enough to ensure it's ability to be scheduled.

Coincidently, guaranteed scheduling also provides a conceptually simpler programming model to the robotic researcher. Rather than having to consider the possible implications of all other tasks running on the system, the programmer can build a behavior in relative isolation, as the behavior concept intended.

In [10], Zhou describes a fast ($O(n \log n)$) dynamic scheduler able to make hard scheduling guarantees. However, her initial multiprocessor implementation only allows a single scheduler to be active at a time. Although actual scheduling overhead is comparatively low, overall scheduling latency could become unacceptable in the presence of a large queue of tasks to be scheduled. A concurrent scheduler would allow multiple schedulers to be active simultaneously thereby increasing throughput and decreasing latency. The design and implementation of a concurrent scheduler is described below.

IV. A Scalable Real-Time Multiprocessor Scheduler

Robot safety requires guaranteed deadlines for tasks but optimal CPU utilization requires dynamic scheduling. These are can be conflicting goals. Below, we describe the design and implementation of the mechanisms and controls that permit multiple distributed schedulers to cooperatively decide task allocation and scheduling. The scheduler uses Zhou's slot list algorithm[10] to perform uniprocessor schedulability analysis, and it employs offers[3] to allocate tasks to processors. Furthermore, scheduling latency is decreased by having multiple processors concurrently perform schedulability analysis for dynamically arriving tasks. Such concurrent schedulability analysis must be performed without increases in uniprocessor task scheduling latency.

These goals determine some basic characteristics of the multiprocessor scheduler.

1. All scheduling information required for uniprocessor schedulability analysis and scheduling must be local to each processor, thereby avoiding increases in uniprocessor scheduling latency.
2. Local slot and task lists must be accessible to remote processors.
3. The task arrival queue, called an *offer queue*, must be shared by all cooperating processors.

Furthermore, contention of access to resources must be minimized. Specifically schedulability analysis must be performed by multiple schedulers such that they do not typically access the same slot list, earliest deadline list, or offer at the same time.

Figure 4 depicts the resulting distributed scheduler. It has five major components:

Offer Queue A shared queue of offers waiting to be scheduled. Each offer describes the execution time characteristics of the offered task. Each offer contains a bid for each processor on which schedulability analysis of the offer succeeds. A *bid* describes the circumstances under which the offered task can be executed on a particular processor.

Slot List(SL) A local list of time intervals occupied on this processor, in time order. Adjacent slots are merged to reduce the time to perform schedulability analysis.

Earliest Deadline List(EDL) A local list of tasks to be executed on this processor, in deadline order.

Scheduler A procedure that accepts locally arriving tasks or tasks in the offer queue, schedules them using the SL, and inserts them in an EDL.

Dispatcher A procedure that removes tasks from the local EDL and executes them on the processor.

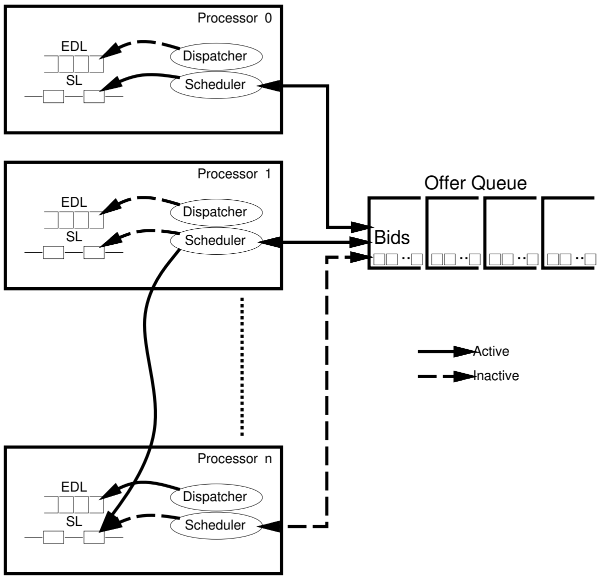

Fig. 4. Structure of the Multiprocessor Scheduler

The multiprocessor scheduler described here is used for dynamic schema scheduling as follows. First the execution of a schema is performed by a thread. Schema scheduling corresponds to the creation and scheduling of a thread executing the schema's code. Thread creation is performed using the call `cthread_fork()`. If the schema is to be executed on the same processor, the thread scheduling analysis is performed by the local scheduler. Otherwise, `cthread_fork()` generates an offer to be placed in the offer queue. Any active scheduler may examine the offer and schedule it using the SL on any processor. When the offer has been scheduled, it is removed from the offer queue. `cthread_fork()` then creates a task, places it in the EDL on the processor where it has been scheduled, and returns to the executing task. Finally, when the dispatcher reaches the task in the EDL, it is removed from the EDL and executed.

A. *cthreadfork()*

```
/* Return TRUE if, before decisiontime, it is
   determined that the amount of time stated
   in runtime can be scheduled on processor node
   between starttime and deadline to execute
   func() on argument arg.  Otherwise return
   FALSE.
*/

RESULT
  RTthread_fork ARGS((int (*func)(),
      any_t arg,
      unsigned int node, /* node mask or list head */
      TIME starttime,
      TIME runtime,
      TIME deadline,
      TIME decisiontime));
```

The result of schedulability analysis is a `bid` data structure. A `bid` is similar in concept to a bid in [8] in that a bid represents the ability of a particular processor to execute a particular task. However, the processes manipulating a `bid` are considerably different. We assume a multiprocessor so that the same algorithm is used for both local and remote scheduling; resource reservation is a parameter of the scheduling policy; scheduling is a three rather than four phase process; and bids may be generated by one processor for another processor.

A `bid` describes the circumstances under which a task can be executed on a particular processor. Upon completion of offer scheduling, an offer will contain a bid for each processor[1] that can execute the function within the given time constraints. `cthread_fork()` will accept the best bid[2] and put a task in the EDL on the processor that generated the best bid. It will reject all other bids.

`Cthread_Fork()` *always* returns on or before `decisiontime`, whether schedulability analysis succeeds or fails. Therefore, the forking procedure can control the duration of analysis. Obviously, a longer `decisiontime` may allow a more complete analysis than a shorter `decisiontime`, and it will affect the likelihood of the fork succeeding. However, due to other time constraints regarding it's execution, a quick decision may be more important to the forking procedure, than an optimal schedulability analysis.

B. *Controlling Contention and Unnecessary Work*

While offers in the offer queue must be accessible by all processors, uncontrolled access could result in very slow processing due to contention between processors for particular memory locations or mutex locks. At the other extreme, restricting access to a single processor unnecessarily eliminates parallel scheduling analysis of an offer that may be run on more than one processor.

The number of schedulers concurrently trying to analyze a particular offer is controlled by a variable, `max_sched_active`, in the offer data structure. The number of schedulers currently trying to schedule this offer is tracked by another offer variable, `sched_active`. When a scheduler is looking for an offer to schedule, if an offer's `max_sched_active` is greater than `sched_active` then the

[1] Subject to `node` and the offer's scheduling policy.

[2] Currently maximum laxity

scheduler will increment `sched_active` and begin scheduling the offer. Otherwise, the scheduler will go to the next offer in the offer queue.

A number of processors concurrently doing scheduling analysis for a particular offer on a particular node achieve no better performance, and probably worse, than a single processor. They are no better because there is only one possible result so they are all doing the same work. They are probably worse, because they will be contending for the same data. Faster offer scheduling requires that different processors schedule different offers or schedule the same offer on different processors.

`Checked` is an offer variable that is bit mask of processors this offer has been analyzed on. `Checked` ensures that cooperating processors analyze an offer on different processors. To minimize contention for the update lock, `checked` is initially tested without locking. If this offer has not been analyzed on one or more processors that this scheduler analyzes for, it is accepted by the scheduler. Later, as the scheduler attempts to analyze the offer on a particular processor, `checked` is locked, tested, updated and unlocked. Since `max_sched_active` limits the number of schedulers concurrently scheduling an offer, contention for the lock is limited. This two phased approach limits both unnecessary work and contention, at the expense of a scheduler sometimes accepting an offer for which it will do no analysis.

V. Scheduler Performance

Our initial experiments used a BBN Butterfly, however that computer has been decommissioned. We are continuing our development on our recently acquired Kendall Square Supercomputer. Specifically, the scheduler performance described below was obtained on our KSR which has a 0.05μsec clock cycle time. These basic performance results demonstrate that the distributed scheduler's mechanisms will deliver suitable performance for the Denning mobile robot discussed in the next section.

offer selection	12μsec
schedulability analysis	165μsec
total offer scheduling	177μsec
mutex lock processing	2μsec

Specifically, on a single processor of a KSR, it takes 177μsec's to select an offer from the offer queue, perform schedulability analysis, and remove the offer from the offer queue.

Figure 5 shows the multiprocessor performance of the scheduler. The single processor times here are approximately 25% longer than the uniprocessor times above because in these experiments the offer list was created on a processor that was not otherwise involved in the experiment. On the KSR, a remote memory access takes four times as long as a local memory access. In a multiprocessor system, it may not be usual to analyze offers on the same processor that generated the offer. In this plot then,

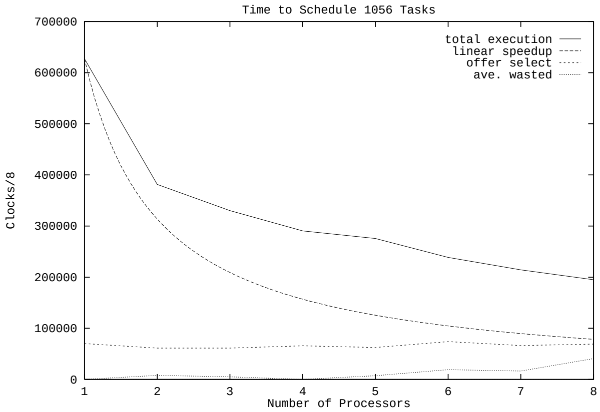

Fig. 5. Multiprocessor Performance

linear speedup is calculated from the execution time on a single processor. It is significant that more processors have only a small affect on the time required to select an offer from the centralized queue. This is because there is no locking associated with selecting an offer. Offers are only flagged when they are selected. At the end of offer analysis mutual exclusion locks ensure that the offer is actually taken by only one processor. This approach does result in some wasted time when two or more processors simultaneously analyze the same offer. However, as indicated by the average wasted time in the figure above, this time is relatively small while there are less than eight processors. This experiment shows that this scheduler design provides good performance for a system of less than eight processors and 100% scheduling overhead. Scalability should improve significantly when tasks are being executed and the percentage of scheduling overhead falls.

VI. Conclusions and Future Work

Dynamic scheduling of schemas offers significant performance enhancement for an example navigational task on uni- and multiprocessors. However, robot safety demands that the scheduler provide guarantees regarding schema deadlines. A prototype concurrent scheduler with run time guarantees has been designed and implemented. Initial data indicates that our approach will provide low-latency scheduling services.

Our goal is to map the control system to a Denning MRV-2 robot at the Georgia Tech Mobile Robot Laboratory. The MRV-2 is a holonomic vehicle with three-wheeled locomotion. It can move up to 4 feet per second. Its primary sensor system is a ring of 24 ultrasonic range sensors, which report ranges to objects up to 200 times per second.

The control system will be restructured to utilize real sensory data rather simulated data. A parallel planner will be included to provide for capability in more complicated situations. The system will make full use of computing resources by dynamically balancing "planner" threads and "motor" threads. In the worst case where robot safety requires the motor threads to monopolize resources, the planner will effectively be paused. If the quality of the existing plan is discovered to be poor, the robot will be slowed. Slowing the robot will reduce the demand motor threads place on the system, thus freeing resources for planning.

The strategy is attractive for several reasons:

- Planning and execution are directly integrated.
- There is no hard and fast schedule; threads are scheduled dynamically as planning needs change.
- Safe speed for the robot is automatically balanced with computational resources.

References

[1] J. Albus, H. McCain, and R Lumia. Nbs standard reference model for telerobot control system architecture (nasrem). NBS Technical Note, Washington, D.C., 1987.

[2] R.C. Arkin. The impact of cybernetics on the design of a mobile robot system: A case study. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1245–1257, Nov/Dec 1990.

[3] Ben Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[4] R. Brooks. A robust layered control system for a mobile robot. *IEEE Jour. of Robotics and Auto.*, RA-2(1):14, 1986.

[5] R.J. Clark, R.C. Arkin, and A. Ram. Learning momentum: On-line performance enhancement for reactive systems. In *IEEE Conf. on Robotics and Automation*, pages 111–116. IEEE, May 1992. Nice, France.

[6] N. Nilsson. Shakey the robot. SRI International Tech. Note 323, 1984.

[7] D. Payton. Internalized plans: A representation for action resources. In P. Maes, editor, *Designing Autonomous Agents*. MIT Press, 1991.

[8] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), 84.

[9] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A c thread library for multiprocessors. Technical Report TR-91/02, Georgia Institute of Technology, Atlanta, GA 30332-0280, January 1991.

[10] Hongyi Zhou, Karsten Schwan, and Ahmed Gheith. The dynamic synchronization of real-time threads for multiprocessor systems. In *Symposium on Experiences with Distributed and Multiprocessor Systems, Newport Beach*, pages 93–107. Usenix, ACM, March. 1992.