Coping with Limited On-Board Memory and Communication Bandwidth in Mobile-Robot Systems

Yilin Zhao, *Member, IEEE,* Chinya V. Ravishankar, and Spencer L. BeMent *Senior Member, IEEE*

***Abstract*—Much effort has gone into studying navigation algorithms for mobile-robot systems. However, although mobile-robot systems often suffer from a lack of adequate on-board memory and communication bandwidth, little work has been done on techniques to solve these problems. Two algorithm-implementation strategies are examined to solve the memory-limitation and communication-bandwidth-limitation problems associated with the navigation of single or multiple robots in large dynamic environments. On-board main-memory-management mechanisms, cache policies, auxiliary-memory data structures, and two path planners are explored by simulations based on a new navigation algorithm. One- and two-level caches with one- and two-level planning, respectively, are investigated; these can easily be extended to schemes with more levels. Our results show that among the seven (three local and four global) cache policies studied, the predicted-window, aisle, and via-point policies overcame the above limitations without compromising robot performance. Therefore, one or more of these three policies can be used with implementation strategies to deal with the memory-limitation and communication-bandwidth-limitation problems encountered in real-world mobile-robot navigation. Our results can also be very useful in the domain of Intelligent Vehicle Highway Systems (IVHS), where the main memory of the on-board computer may be too small to hold all of the road network and other useful information.**

I. Introduction

LARGE dynamic environments pose great challenges for single or multiple mobile-robot navigation systems. These challenges are both at the level of algorithms and in their implementation. In this paper, we address the latter set of challenges by studying strategies for implementing these algorithms under hardware constraints.

When mobile robots travel in very large environments, the amount of information they must access is typically very large. As an example, our experimental robot needs a 128 × 128 array to represent a small 12.8 × 12.8 m^2 indoor test field. Since memory requirements tend to increase as the square of the field size, an outdoor environment would need much more memory. Although main memory sizes have been increasing, they are still not large enough to contain very large data sets. An alternative must be used; that is, the data must be stored in an auxiliary memory instead. Similar difficulties arise in the newly developing field of Intelligent Vehicle Highway Systems (IVHS). In order to find the best route connecting the origin and destination points for a vehicle over a road network, the system must store in memory the details of the road network. Typically, at least 30 Mbytes are required for a metropolitan area, such as Chicago or Detroit. Therefore, an auxiliary memory is commonly used [16].

This situation raises some problems. First, access to the auxiliary memory is typically several orders of magnitude slower than access to main memory. Therefore, how to access data from the auxiliary memories is an issue critical to the success of this approach. One technique to address this memory-management problem is the *cache*.

A cache is a collection of data blocks that logically belong on the auxiliary memory, but which are being kept in the on-board memory for performance reasons. In other words, a cache provides a high-speed buffer in the on-board main memory to act as a temporary store for the auxiliary memory. Caching is an old technique, and cache memories have been used in computers for a long time. Relevant references on this topic are [5], [12]–[14]. In this paper, we do not study the various aspects of cache memory design, or their implementation policies for general-purpose high-speed computers. For instance, we will not examine cache design issues such as the optimal cache size [13], block (line) size [14], degree of associativity [5], and whether to update memory by write-though or copy-back approaches [12], [13]. We focus instead on how to use cache memories to implement algorithms for mobile-robot navigation, and on the effects of the proposed implementation strategies for mobile-robot performance.

We intend to use these strategies and associated cache policies to solve the memory-limitation and communication-bandwidth-limitation problems which we have described. While many caching techniques exist for general-purpose computers, we believe they are quite inappropriate for our application domain.

A. Caching Issues

There are several reasons why general-purpose schemes are unsuitable for our purposes. First, a general caching policy deals with both instruction and data caches. In our domain, code access patterns are thoroughly predictable, but data management poses great challenges. A commonly used cache replacement policy for general-purpose

Manuscript received March 6, 1992; revised February 17, 1993 and April 21, 1993.

Y. Zhao was with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109. He is now with Motorola Inc., Northbrook, IL 60062.

C. V. Ravishankar and S. L. BeMent are with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9212924.

computer systems is least recently used (LRU). Whenever the CPU needs a block of data which is not in the cache, it will fetch that block from a disk and replace the one in the cache which is least recently used.

Now, the LRU idea assumes that the locality patterns need not be precisely known, but that they are captured well by the LRU technique. In our case, we know much more about the data access patterns. As discussed in the next section, the navigation code is often focused strongly on one data window at a time. Also, the LRU technique focuses on how to replace blocks of data in the cache as new blocks are brought in. In our domain, we are concerned less with where to put new data, and much more with what data will be accessed next. In other words, in a classical LRU algorithm, a miss brings in a new block of data, the extent and definition of which is determined without pro-actively considering program semantics or characteristics. The same fetch/replacement strategy may be used for both mobile-robot applications and for supercomputing applications even though these applications have different characteristics. In contrast, we concentrate particularly on the issue of navigation and on relevant data.

Furthermore, in our domain there are time-critical routines in the robot control system. If the CPU is forced to fetch a block of data during the execution of a time-critical routine, the system may miss deadlines, leading to a dangerous accident. This issue is discussed in greater detail in Sections III-A and III-C.

We have observed, in particular, that using a general-purpose LRU-style algorithm for global path planning can lead to a disastrous situation called *thrashing*, where the system is constantly demanding access to blocks which have been swapped out to disk by the LRU algorithm. This situation creates enormous traffic between the main store and auxiliary store, and the system performance degrades severely. For example, a best-first heuristic-search algorithm used for global path planning must find a path along the one branch that is the best among the rest of the branches in the search tree, so it often jumps from one branch to another during execution. As the algorithm searches a new branch, the blocks of data brought for the previous branch tend to get swapped out of the cache. If the algorithm jumps back to the previous branch, it must refetch the data surrounding this branch. These data must be fetched from the auxiliary memory, creating tremendous traffic between the cache and auxiliary memory, and forcing the CPU to wait. In an actual case, it was observed that using a best-first search for planning with an LRU-based cache caused the CPU to idle 50% of the time waiting for the caching algorithm to fetch data from auxiliary memory. Therefore, an in-depth study of different implementation strategies is clearly necessary.

A look at the literature in the mobile-robot field [8], [17] suggests that much attention has been paid to developing new navigation algorithms, and to extracting and representing different sorts of sensory information for navigation. On the other hand, we have seen no studies which deal with the challenge of how to implement algorithms with multilevel map representations for single- or multi-robot navigation in large environments with memory and communication bandwidth limitations. We address these problems here in the context of the navigation algorithm developed in [20]. It is generally agreed that mobile-robot navigation algorithms should be based on multiple levels of spatial representation or global–local map models. This navigation algorithm (Appendix A) uses a node-based map for global planning and a grid-based map for local planning, which is very similar to the hierarchical representation proposed by other researchers. Therefore, we anticipate that our studies will provide information of direct use to others in the field.

We propose one-level and two-level cache mechanisms to solve the memory-limitation and communication-bandwidth-limitation problems. Only portions of local and global map data are stored in the main memory of an on-board computer, with the rest of the data being stored in an auxiliary memory. The auxiliary store may either belong to one robot or be shared by several robots, and may either be a disk or a large off-line main store. Whenever the required data are not in the on-board memory, they are brought in from the auxiliary store. This mechanism is useful not only for the case where on-board memory is insufficient, but also for low-budget projects which cannot afford to upgrade to large-memory computer systems. Our results can also be very useful in the domain of Intelligent Vehicle Highway Systems (IVHS), where the on-board computer's main memory for a vehicle cannot hold all of the road-network information, which is so large that CD-ROM compact disks are required for the auxiliary memory [16].

We have constructed a simulator to study the proposed mechanisms using our navigation algorithm. For multi-robot navigation, we assume a distributed approach. That is, each robot in the field uses its own planning algorithm and views all other robots as moving obstacles. Researchers in distributed artificial intelligence often take a different approach, e.g., [3], in which robots are not passive obstacles, but participants in a cooperative problem-solving endeavor. Since the emphasis in our study is on memory management, we will limit discussion of such algorithmic and strategic issues. Furthermore, our two-level cache mechanism can be extended to a multilevel cache mechanism if a multi-layer map is used for navigation. We will introduce these implementation strategies, and provide simulation results of these strategies and associated cache policies.

Our results suggest that the recommended cache policies and suggested implementations overcome the on-board memory and communication-bandwidth limitations without compromising robot performance. Therefore, they can be used to deal with memory-limitation and communication-bandwidth-limitation problems. There are many aspects of the multirobot navigation which are not handled well by the LRU-like general cache schemes. Our work addresses these issues and provides new mecha-

nisms for the memory management of multirobot navigation in large environments.

II. Implementation Strategies for Large Dynamic Environments

One of the main problems in the autonomous navigation of mobile robots is coping with large volumes of dynamic environmental information when the on-board memory and communication bandwidth are limited. Two algorithm-implementation strategies are proposed and discussed in detail.

A. Problem Formulation

For this study, we choose a scenario where one or more robots work on the same floor of either a big warehouse or an environment with many random obstacles. The warehouse scenario can be extended to supermarkets, busy hospitals, etc. The environment is dynamically updated whenever aisles become blocked or opened in the warehouse. Furthermore, we assume that the main memory of the on-board computer is too small to store all the environmental data for the robot. Therefore, each robot will need to communicate through buses, radio links, or other transmission carriers to obtain the rest of the data from an auxiliary memory, which could either be a disk or main memory in an off-line central control unit. This memory is either owned by one robot exclusively, or is shared and accessed by several robots through transmission carriers. Each robot need not hold all the environmental information when a pool of memory is shared by several robots. Sharing memory also frees up memory for other important tasks.

A robot not only detects obstacles dynamically with its sensors, but also uses the relatively static environmental information preregistered in the auxiliary memory. This preregistered information will help filter out the errors caused by sensor inaccuracies, and also help to estimate robot position, so crucial to navigating robots in large environments. Because a dead-reckoning position system accumulates errors as the robot travels and an external-sensor position system requires modifications to the environment, it is becoming common to use both preregistered and sensor-detected environmental information to correct the errors accumulated by a dead-reckoning system [2], [9]. Therefore, both sensor-detected environmental information and known map data are assumed as important for navigation.

We propose two strategies to address memory-limitation problems. First, if only one robot is operating in a relatively static environment, a local path planner could be used. Since a local path planner uses only the local data immediately surrounding the robot, it is clear that only a very small portion of the on-board memory is needed for this planner. The rest of the memory can be released for other robot functions, for example, for a vision system. This strategy can also be used for environments where traps can occur, as discussed in [18], [20]. However, this case does not represent fully autonomous navigation, because the robot must wait either for the obstacles to be removed or for a human operator to reset the robot. We call this strategy and the associated memory-management mechanism to be introduced the *local strategy*.

If one or more robots work in an area where the environmental information is updated very frequently, hierarchical integration of local and global path planners would be a good choice. In this integration strategy, a detailed local map is used by the local path planner, and a global, more abstract map is used by the global path planner. Clearly, the memory-limitation problem is worse now because it is less likely that all of the required information could be stored in the main memory of the on-board computer. We call this integration strategy and the associated memory-management mechanism described below the *global strategy*.

A memory-management mechanism must be devised to solve the on-board memory-limitation problems for both the local and the global strategies. In order to implement our proposed mechanism we use caching to manage on-board memory, as depicted in Fig. 1. For the local strategy, only a local data cache is needed. For the global strategy, both global and local caches are required. We assume that when any robot finds new information, the relevant local and global maps will be updated. Based on research results on distributed cache consistency [10], [15], we can safely assume that our map-data caches are also consistent.

We have studied the effects of various caching policies for robots working in large environments through simulation. The strategies considered here can be implemented with any local and global path planners, but the results obtained will vary. Our study uses a potential-field-based local path planner and a heuristic-search-based global path planner [20].

B. Navigation with a Local Data Cache: Local Strategy

In this subsection we obtain two criteria for guiding local path planner design and discuss three local-cache policies.

1) Design Specification: By definition, a local path planner uses local data for its planning process. This planner must acquire environmental data surrounding the robot in real time. In our system, this segment of data is called an *active window* W, and is limited by the sensor viewing distance. Fig. 2 shows two active windows W_{t_i} and $W_{t_{i+1}}$ used by the local path planner at sampling times t_i and t_{i+1}, respectively. The center of the robot is indicated by the small circle in the center of the active window $W_{t_{i+1}}$.

We now obtain some simple analytical estimates of the possible efficacy of caching. To simplify, we omit the justification of the following criteria and associated examples, which can be found in [17]. First, we note that in a successful obstacle-avoidance algorithm, the active window must be used in accordance with the following criterion:

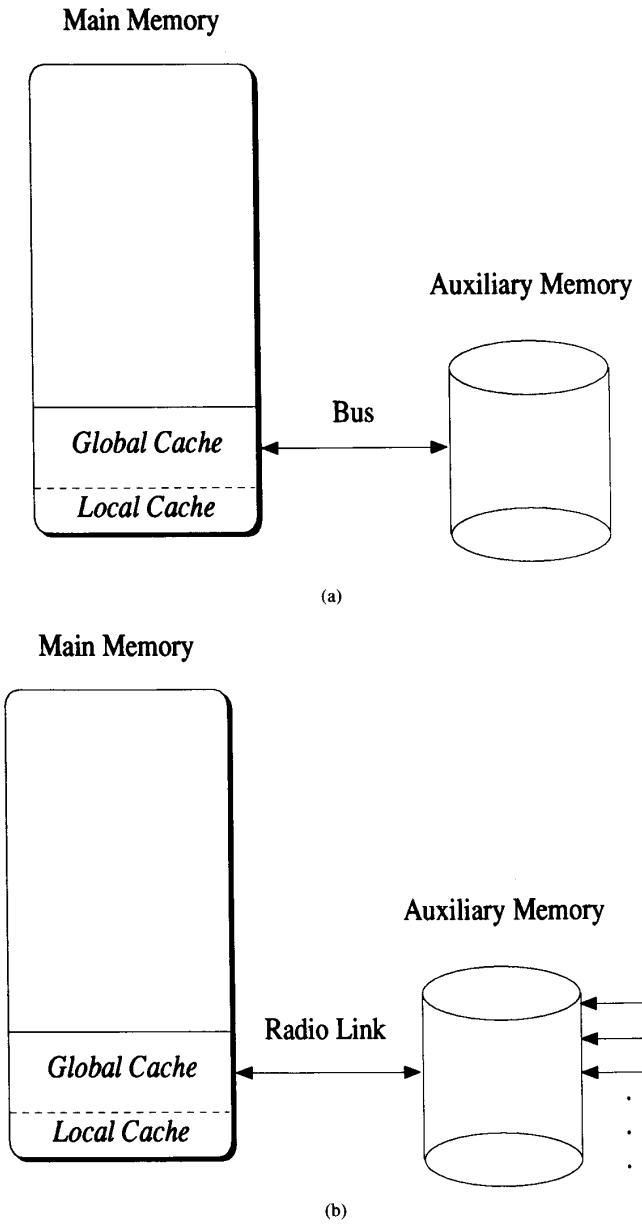


Fig. 1. Two-level cache for navigation. (a) Single-robot navigation. (b) Multirobot navigation.

Criterion 1: For safe maneuver, the new active window must overlap the previous active window, so that the current location of the vehicle is included in the previous active window.

If this criterion is violated, there will a time period when the robot moves without knowing exactly what surrounds it. On the other hand, this criterion implies that if we can cache the entire active window into on-board main memory (local cache), many elements or cells used for the previous window are still useful for the next window, and do not need to be brought in from the auxiliary memory. The miss rate in the cache is lowered.

Criterion 2: For a successful local caching policy, the transmission rate between the main memory and the auxiliary memory (or a shared memory) should satisfy (1):

$$\hat{a} \geq p \frac{(2l)bv}{\sqrt{2}\, d_u} + (1 - p) \frac{lbv}{d_u} \tag{1}$$

where p is the probability of diagonal updating for the active window attached to the robot, l is the active window dimension, b is the number of bytes stored in the local map for each element, and a (bytes/second) is the expected or average transmission rate required for a successful cache.

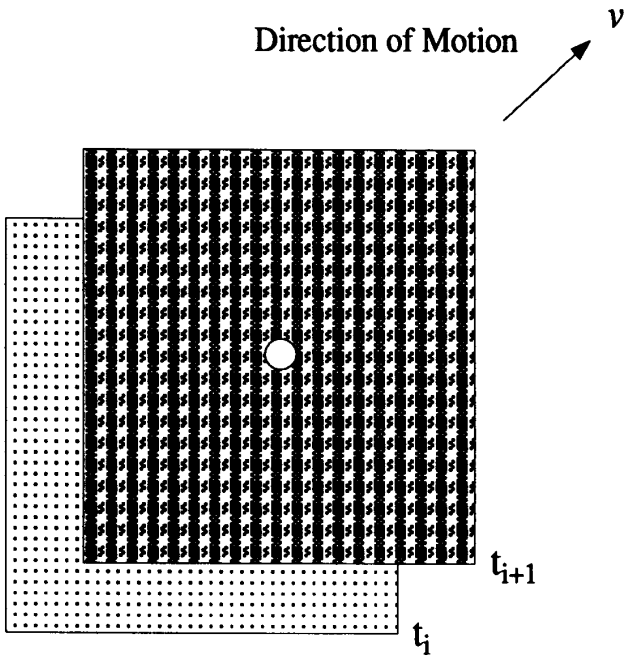


Fig. 2. Active windows for local path planners.

The robot itself can move in any direction. However, because of the integer indexing of the active window array, this window can only be updated horizontally, vertically, or diagonally. The first term in (1) is the case where the active window is updated diagonally. The next term corresponds to the situation where the active window is updated either horizontally or vertically.

Similarly, we can estimate the *active-window reference count* which is the number of times an active window is continuously used.

$$\hat{\gamma} = p \frac{\sqrt{2}\, d_u}{vT_1} + (1 - p) \frac{d_u}{vT_1} \tag{2}$$

where d_u is the unit distance between adjacent cells of the array which stores the local data, T_1 is the sampling period[1] of the navigation algorithm, v is the robot velocity, and p is the probability of diagonal updating for the active window. For our experimental CARMEL robot,[2] the maximum velocity is 0.78 m/s and the sampling period is 0.03 s. It turns out that the reference accounts for our robot are between 6.04 and 9.43 [17].

We have found that even with a relatively fast vehicle, the active-window reference count is greater than 1, which means that the same window will be used repeatedly by the vehicle. This result indicates that caching is likely to be effective.

[1]The sampling period is the time period required to issue two consecutive speed and steering commands and to acquire the vehicle position and orientation values for the system state during navigation.

[2]CARMEL is a modified commercially available CYBERMATION K2A mobile platform.

2) Local Cache Policies: In the following discussion, we assume that the buses, radio links or other transmission carriers are fast enough to cache the required elements without significant delay. Some implementations need to prefetch blocks of elements from the auxiliary memory into the cache. We assume that this prefetching will not cause noticeable delay in the planning process or the sampling period. How to deal with the delay problem (*cache latency*) using proper data structures for auxiliary memories can be found in [17]. For the local cache, we consider the following three policies:

When a cache miss[3] occurs (or before it occurs),

- L_1: bring the exact active window into the cache (exact-window cache),
- L_2: bring a predicted active window into the cache (predicted-window cache), or
- L_3: bring one element into the cache (one-element cache), which is equivalent to no caching.

The exact-window cache (L_1) is impractical. It is impossible in practice to bring in the exact active window whenever a miss occurs, since it is possible neither to know the exact window nor to bring it in immediately. Therefore, this policy is presented solely for the purpose of comparison. When a missed element is identified, the system will bring in an active window which contains the missed element.

For the predicted-window cache (L_2), two storage segments are reserved in the cache. One segment contains the

[3]A cache miss occurs when the expected element is not in the cache, so that fetching from an auxiliary memory is necessary.

current active window and the other is used as a work area to bring in a predicted active window. That is, while the system is using one active window, another window is being brought in. When the navigation algorithm is finished with the currently active window, it expects to find the next window in the cache. Criterion 1 requires the current active window to overlap the previous one. Thus, it turns out that there is no need to allocate two complete active windows in the cache. Theoretically, if the predicted window always contains what the system needs, the cache hit ratio will be 100%.

The navigation algorithm accesses map data elements in the on-board memory. We call the number of times the memory is referenced the *memory reference count*. The number of times a referenced element is found in the cache is called the *hit count*. The ratio of the hit count to the memory reference count is called the *hit ratio*.

Different methods can be used to predict the next active window. Since estimating the next active window is equivalent to estimating the next location of the robot, we propose modifying the equation derived in [18] for this task, as described in Appendix B.

For the one-element cache (L_3), only one element will be brought in whenever there is a miss in the cache. Compared with the previous two policies, this one should have the lowest hit ratio, unless the second policy makes particularly bad predictions. A one-element cache may be the only option when the available memory is too small to allow prefetching. At any rate, it serves as another extreme for purposes of comparison.

The following discussions indicate that the predicted-window cache (L_2) is better than the one-element cache (L_3) for dealing with communication-bandwidth limitation problems. Because the speed of the robot is low enough, the active-window reference count is always greater than one. This means that we have more than one sampling period to fetch the required elements from the auxiliary memory.

From our earlier estimates of the active-window reference counts (6.04 to 9.43), we can estimate that approximately 180 ms to 300 ms ($\hat{\gamma} \times T_1$) are available for our robot to fetch new elements. This time is insufficient for a one-element cache which must fetch elements one by one. Assume that the auxiliary memory is a disk. Disk access time includes seek time, rotational latency, and data transfer time. We use the disk access time estimates of Katz *et al.* [7], who have studied the disk performance characteristics of various classes of applications. Since our application is a scientific application in their classification, we obtain the average seek time as 15 ms, the average rotational latency as 8 ms, and the average data transfer time as 15 ms. Therefore, the average data fetch time is 38 ms. Even in the worst case, the next window never differs from the present window by more than one new row and one new column. Therefore, for the one-element cache, in the worst case we must fetch at most 65 elements for a new window (for one new row and one new column). Thus, the one-element cache needs 2.47 s (38 ms $\times$ 65) to fetch all these elements. Even if we assume only two rotational latencies, one for the row elements and another for the column elements, the fetch time still turns out to be around 1.966 s (30 ms $\times$ 65 + 8 ms $\times$ 2). This time is far beyond the available time (from 180 ms to 300 ms).

For the predicted-window cache (L_2), if we can fetch the row and the column separately as two blocks of elements, the fetch time is only 76 ms (38 ms $\times$ 2), which is well below the required available time period. Therefore, for a system of low communication bandwidth, the predicted-window cache is a better choice. However, for the predicted-window cache to work as requested, we must find a good data structure to allow row access and column access with the same efficiency. How to store the data in the auxiliary memory to reduce the cache latency can be found in [17].

C. Navigation with Local and Global Data Caches: Global Strategy

In this subsection, we examine methods for the invocation of global planners. Four global-cache policies are also discussed.

1) Global Path Planner Triggering: A global path planner may be triggered in two ways: 1) whenever the global data are updated, and 2) whenever the robot becomes trapped. We use the second trigger in our simulations. When several robots are working in the same environment, the global map is updated as each robot detects new information. However, a robot will not need to replan its path at this point unless the planned path becomes blocked. Therefore, when the global map is updated, the cache process, not the global path planner, will be activated if the corresponding elements are in the cache. We thus eliminate unnecessary invocations of the global path planner by implementing the second trigger.

2) Global Cache Policies: We examine four global cache policies:

- G_1: hold a fixed portion of the global data in the cache (fixed cache),
- G_2: hold all the global data along the path and as much surrounding data as the cache allows (path cache),
- G_3: hold only aisle data[4] in the cache (aisle cache), or
- G_4: hold the global data along the preplanned path between two via points[5] in the cache (via-point cache).

A fixed cache (G_1) holds a predetermined portion of the data. This unsophisticated policy is used as a basis of comparison. For instance, if only half of the global data is permitted in the cache because of global cache memory limitations, we could hold the portion of data which represents the lower half of the global map in the cache. Of

[4]The aisle data are data that represent the free aisles or hallways in the environment. In the cases where there are no clearly defined aisles, the aisle data represent the free space.

[5]Via points are intermediate points (or goals) on the globally planned path for use by the local path planner to reach the final destination.

course, with slight modifications, different portions of the map data can be in the cache. One such choice is described in the second policy.

A path cache (G_2) holds the data along the planned path together with some data around the path to permit path replanning. The amount of this extra cached data depends on the global cache memory size. To start with, the algorithm could permit only that portion of data immediately adjacent to the starting position in the cache, or the data along the straight line from the starting position to an intermediate target position in the cache. Whenever a new path is planned, the data along this new path will be cached in.

An aisle cache (G_3) holds only the data representing the aisles. This policy is very useful for an environment where the aisle area is far less than the area occupied by obstacles, since there is no need to cache in a large amount of extraneous data. Many warehouse and building floors are examples of this situation.

A via-point cache (G_4) will only hold data in a *planning region* that covers a number of via points on the path, including the current via point, together with some data around these via points. This policy is a combination of the fixed and path caches, and should be useful for environments with random obstacles. To use this policy, we must restrict on-line global planning to look at only this planning region, whose size is determined by the capacity of the global cache. Fig. 3 shows two simple planning regions, which are in the cache at time t_j and t_{j+1}, respectively. These regions are portions of the global map and are shown as dashed and solid parallelograms. Small squares represent via points.

Using Fig. 3 as an example, one way to implement this policy is to cache all data between two via points and include some surrounding elements, or to start with, use the approach suggested for the path cache (G_2). At time t_j all the elements inside the dashed region are in the cache. The first via point in the cache is the one at the bottom of the parallelogram, and the last one is on the upper boundary. The number of elements cached in depends on the global cache size selected. As the robot travels, the system will update the global cache contents. Whenever the vehicle passes a via point, it will refresh the global cache with the via point following the last cached-in via point, and eliminate the via point which the vehicle has just passed. Some surrounding elements must also be cached to facilitate replanning which requires surrounding context. Otherwise, if one via point is invalid, a high hit ratio cannot be achieved for replanning using only these remaining via points in the cache. Based on the real sensor viewing distance restriction, a pair of via points can be no more than 2 m apart in our navigation algorithm [20], and our robot travels that distance in about 4 s. Because each pair of via points is five nodes apart in the search graph, no more than five row elements from the global map are needed to refresh the cache. Thus, the global cache has 0.8 s to fetch each row from the global data array, which should be enough time if a disk is used as the auxiliary memory. During the journey, once the global path planner is invoked, it will use its last cached-in via point as a temporary target to replan the new path.

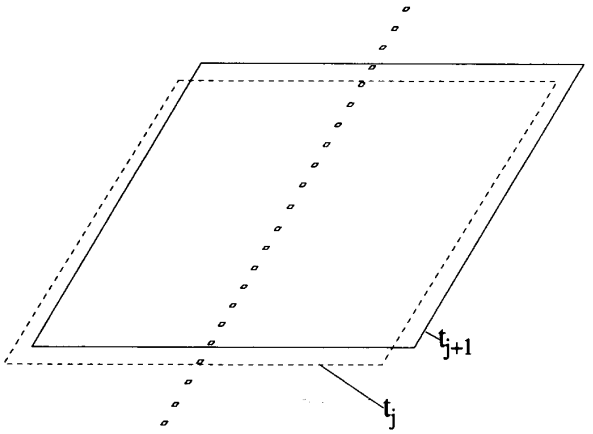


Fig. 3. Via-point cache (G_4).

D. Discussion

Locality is the most important property for guaranteeing the successful use of cache memories [12]. Locality has two characteristics: locality by time and locality by space. Locality by time means that the information which will be in use in the near future is likely to be in use already. Locality by space means that portions of the address space which are in use generally consist of a small number of individually contiguous segments of that address space.

For our local cache, locality is guaranteed because the continuous motion of the vehicle requires the use of adjacent windows from a local map. The three local cache policies will not violate the locality property. For the global cache, locality is less likely to be guaranteed because of the nature of the global path planning algorithm. The elements of the global map array will most likely be randomly accessed during planning. Therefore, we expect that the fixed cache (G_1) and the path cache (G_2) are poor policies in terms of the locality criterion. These policies cannot guarantee locality by time and locality by space because it is difficult for them to predict the elements accessed by the heuristic search. However, we still include them in later simulation studies to evaluate the efficacy of the fixed cache G_2 and the path cache G_2. Since the aisle cache (G_3) and the via-point cache (G_4) try to place the related elements for global planning in the cache, they should perform better than policies 1 and 2, based on the locality criterion. Details appear in Section III-B.

III. Simulation Results

Our cache-based planning simulator is constructed with the software actually used in our experimental mobile robots. The local and global path planners are both briefly described in Appendix A and presented in detail in [20]. The first is potential-field based and the second heuristic-

search based. Taking the IBM-compatible CPU used in our robot as a prototype, we assume that only 1 Mbyte of main memory is available, of which only 200 kbytes can be used for both local and global caches. Under this restriction, we try to determine which policy performs best. We then change the cache size to see the effects of cache on the hit radio.

A 1024 × 1024 array is used for detailed local data stored in the auxiliary memory. Each cell in this array represents a 10 × 10 cm^2 floor area in the environment. A 256 × 256 array is used to store global data that coarsely represent the environment, and which are also sorted in the auxiliary memory. Each local element occupies 1 byte and each global elements occupies 6 bytes. The active window size is 33 × 33. Therefore, if we place the current active window into the cache, only half of the global data can be cached into the global cache, assuming that only 200 kbytes are available for the local and global caches.

To test our policies, we simulated a warehouse environment for the first three experiments. Fig. 4 shows this warehouse, 102.4 m × 102.4 m square. Each aisle is 2 m wide. We assume that the robot begins its journey at the lower left corner, marked "S" and navigates to its final destination at the upper right corner, marked "T." New obstacles within sensor viewing distance of this and other robots are added dynamically to the local and global maps.

A. Test of Local Strategy

In this experiment, we tested three local cache policies for the first strategy, which uses only a local data cache. The robot traveled with the guidance of only a local path planner. The final trajectory of the robot is shown in Fig. 4 as a continuous curve.

The results from the three policies are listed in Table I. The cache misses for the exact-window cache (L_1), the predicted-window cache (L_2), and the one-element cache (L_3) are 2135, 23685 and 69427 elements, respectively. All of these numbers are small compared to the hit count, so that every policy produced more than a 99% hit ratio, even for a one-element cache (L_3). These results are expected because of the very high active-window reference count. This active-window reference count can be estimated from (2).

From these results, we conclude that a local path planner can be used in a very large environment even when there are practical limitations on the on-board memory and communication bandwidth, provided the local cache size is not smaller than the active-window size (augmented by a row and column for a predicted-window cache). Since only local planning is used, the problem with this strategy is that the robot can be trapped when the environment is dynamic. When this occurs, the alternatives are to wait until the obstacle is removed or the robot is redirected by an operator.

When the robot speed is increased, more data must be fetched, affecting the hit ratios. The results of using a high-speed robot or long sampling periods with a local path planner are shown in Table II. The second column gives the results for a robot running five times as fast as our experimental robot, i.e., maximum velocity 3.9 m/s, a likely speed for cross-country vehicles. In columns three and four, the results for sampling periods of 300 and 600 ms are shown. These cases could arise even when the robot travels at normal velocity (maximum velocity 0.78 m/s). For example, when a cache miss occurs, the system may need more time than previously expected to bring in required elements from the auxiliary memory. Alternatively, as a project develops, other navigation functions may be added to the navigation process, forcing the sam-

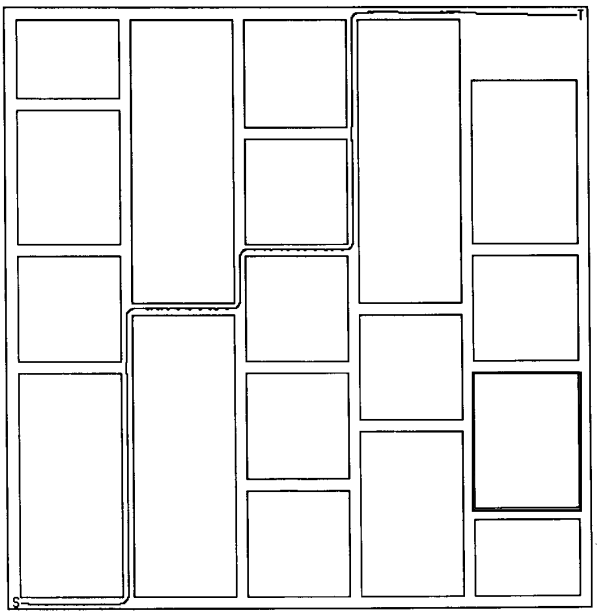


Fig. 4. A warehouse environment and a robot journey under a local path planner.

TABLE I
LOCAL CACHE POLICY COMPARISON

Policies(v = 0.78 m/s, T = 30 ms)	Hit Ratio	Hit Count	Memory Reference Count
Exact-Window Cache	99.98%	9 495 194	9 497 329
Predicted-Window Cache	99.75%	9 473 644	9 497 329
One-Element Cache	99.27%	9 427 902	9 497 329

TABLE II
EFFECT OF VELOCITY AND SAMPLING PERIOD

	Hit Ratio		
Policies	30 ms (T) 3.9 m/s (v)	300 ms (T) 0.78 m/s (v)	600 ms (T) 0.78 m/s (v)
Exact-Window Cache	99.92%	99.90%	99.90%
Predicted-Window Cache	97.86%	95.12%	92.12%
One-Element Cache	97.18%	92.16%	84.89%

pling time to be longer. Our simulation results indicate that the hit ratio gets worse when either the robot speed or the sampling period is increased. In particular, the hit ratio drops considerably for the one-element cache (L_3) when the sampling period is 600 ms.

Fig. 5 shows the hit ratios for robot speeds between 0.78 and 3.90 m/s. In this test, the predicted-window cache (L_2) was used. The hit ratio begins to decrease relatively rapidly as the robot speed increases beyond 1.5 m/s.

1) Effects on Robot Performance: The hit ratio explicitly reflects the performance of the local cache. It also implicitly reveals several problems which robot designers and users face. Because the exact-window cache (L_1) is used for comparison purposes only, we examine the results for the predicted-window cache (L_2) and the one-element cache (L_3) in Table I. Assume that the auxiliary memory is a disk and the transmission carrier is a bus.

We first consider the time required to update one window. As discussed in Section II-B, fetching a single block of elements for the predicted-window cache takes only 38 ms if row-data fetching and column-data fetching are made equivalent. We define *cache latency* to be the time taken to fetch the missed elements, so 38 ms is the cache latency. In this particular warehouse example, there is almost no diagonal updating of the active-window because the vehicle moves either horizontally or vertically with respect to the world coordinate system. Thus, we assume that one fetch is enough to update the active window (for two fetches, the calculations below can be adjusted accordingly).

We then consider the effects of the predicted-window cache and one-element cache on robot navigation process. For the predicted-window cache, the 99.75% hit ratio means that 0.25% of all memory references result in misses. Since the cache latency is 38 ms for this case, the navigation process will be delayed 38 ms when a miss occurs. For the one-element cache, the maximum number of elements we must fetch is the window width, i.e., 33 elements. As discussed before, the disk-seek and data-transfer take 30 ms. Assuming only one rotation latency (8 ms) and a cache block size of one element, 33 fetches are required during one sampling period. Thus, even in the best case, the navigation process will take about 1 s (33 × 30 ms + 8 ms) to update the window (33 elements). For the one-element cache, the 99.27% hit ratio means that 0.73% of all memory references result in misses, based on counting every single missed element. Comparing the 1 s cache latency for the one-element cache with the 38 ms cache latency for the predicted-window cache, we see that the predicted-window cache is much better. A one-second cache latency is much longer than the navigation process can afford (from 180 to 300 ms), so the robot must pause to satisfy the cache latency-requirement and Criterion 1 for the one-element cache. In contrast, for the predicted-window cache, the robot needs only to slow down briefly to compensate for the 38 ms delay.

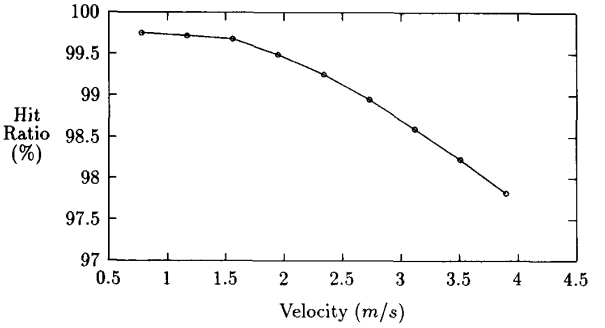


Fig. 5. Effect of velocity with predicted-window cache (L_2).

Finally, we consider the effects of cache misses on robot performance. For most applications, slowing down to compensate for the cache latency is unnecessary if the predicted-window cache (L_2) is used. We can initially ignore the missing elements. The robot misses at most 33 elements of the current active window, i.e., 3% of the total window elements. Since our local navigation algorithm is potential-field based, elements close to the robot position will have a far greater influence than those farther away (see, for example, [1] and [20]). The 33 elements that were missed lie on the boundaries of the active window, and have little influence on the navigation process. Because the active-window reference count is always greater than one, the same active window is used again and again, so this missed row or column will soon be available. However, for the one-element cache (L_3), missed elements (at least 33 elements for one sampling period) cannot be ignored, since this will violate Criterion 1. After a short period the local cache will lose all of the window information related to the current location of the robot. Clearly this cannot be tolerated for safe robot maneuvering. Therefore, the one-element cache is not an acceptable policy.

B. Test of Global Strategy

With the two-level hierarchical planner implemented in our simulator, the robot was able to perform path maneuvers without human intervention. Three different situations were tested in this study. First, fixed obstacles were placed on the warehouse floor to try to block the path of the robot. Second, random obstacles were placed on the warehouse floor. Finally, in a dynamic simulation fixed numbers of obstacles were randomly placed on an empty floor and obstacles were then removed and added randomly, keeping the total number of obstacles fixed. The predicted-window cache (L_2) was used for the local cache, while different global policies were studied for the global cache.

1) Manually Updated Warehouse Environment: The robot first used the global path planner to plan its path from the initial position to the final target, as indicated by the small squares in Fig. 6. Immediately after the completion of the initial global path planning phase, an obstacle was added to block the intended path of the robot (in-

dicated by arrow 1 in Fig. 6). The robot moved along the planned path, as indicated in the figure, until it encountered the first obstacle, wherefrom it planned a new path, backed out of the blocked aisle, and continued along a new path. Again this new path was blocked (arrow 2) and the robot replanned and continued until either no obstacle was in its way to the target or there was no way to reach the final destination.

From Fig. 6, we can see that the robot finished its journey after planning the global path four times. The results for the three different policies for the global cache are listed in Table III. It is confirmed here that the fixed cache (G_1) and the path cache (G_2) perform poorly, in accordance with our earlier expectations. The results of the three policies in the table show that the aisle cache (G_3) is the only good choice in this case. The other two policies would result in much lower hit ratios.

We now consider what these hit ratios tell us. Besides the high miss ration (1 − hit ratio), the cache latencies of the fixed cache (G_1) and the path cache (G_2) are also high. Assume that the auxiliary memory is a disk and that the cache block size is equivalent to the global data element size. From the discussion in Section III-A, we know that the cache latency for each element is 38 ms.

Our global planner uses a best-first strategy. This technique can result in unpredictable jumps over the search space, and we model this situation as causing random accesses to elements. The total cache latency thus is obtained by multiplying the number of misses by the one-element cache latency. For the fixed cache (G_1), this cache latency works out to be 23.76 min. Since the global path planner is invoked four times, the average cache latency for each on-line global planning is 5.9 min. For the path cache, the total cache latency is 15.26 min. The average cache latency for each on-line global planning is 3.8 min. Since the global path planner is invoked four times, the robot will spend an extra 15.26 min to reach a final destination that would have been reached in less than 10 min with no cache latency.

In this particular experiment, only 12 105 elements are required in the cache for the aisle cache (G_3), while for the fixed cache (G_1) and the path cache (G_2), 32 768 elements are required. Although only a small amount of data is in the cache for the aisle cache, some cache activity is still expected. For instance, newly discovered obstacles may require changes to elements currently in the cache. Because only three obstacles were added, 26, 78, and 66 elements were dynamically cached for the fixed, path and aisle caches, respectively, in addition to regular cache activities. We call them *additional elements fetched*. Since only a fixed portion of the data is in cache for the fixed cache, not all newly detected obstacles required changes to elements in the cache. The fixed cache thus has the fewest additional elements fetched, even though obstacles are placed in the aisles. Our heuristic-search algorithm always expands the obstacle boundaries because of security considerations [20]. Because the aisle cache (G_3) caches only aisle data and does not include some of the expanding boundaries surrounding newly added obstacles, the number of additional elements fetched is more than for the other two schemes.

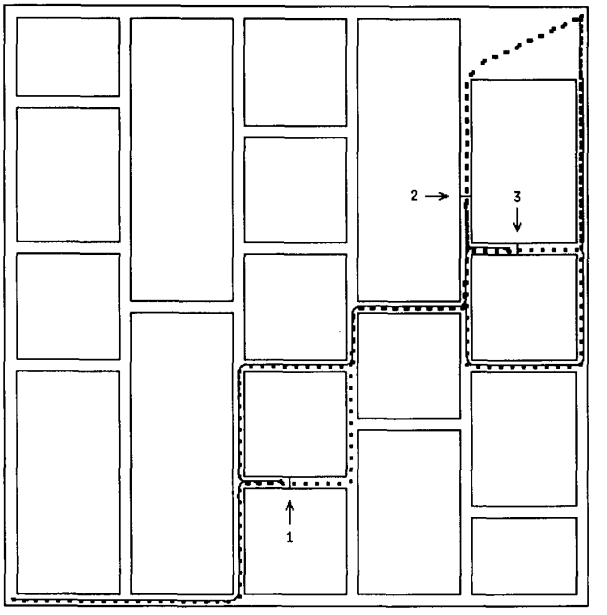


Fig. 6. Navigation with manually updated obstacles.

TABLE III
GLOBAL CACHE POLICIES WITH FIXED OBSTACLES

Policies	Hit Ratio	Hit Count	Memory Reference Count
Fixed Cache	44.09%	29 584	67 092
Path Cache	64.08%	42 995	67 092
Aisle Cache	100.00%	67 092	67 092

We see that the hit ratios of the fixed cache and the path cache are significantly worse than for the aisle cache. Although the memory reference count of the global cache is only 0.456% of the total memory reference count, the misses will significantly affect the performance of the system, as we learned above. The robot must halt to compensate for the long cache latency.

In Fig. 7, we show how the cache size affects the hit ratio of the aisle cache (G_3). The hit ratio drops rapidly as the cache size is reduced below 12 105 elements since the global cache can no longer hold the entire aisle data. Therefore, below about twelve hundred elements the hit ratio decreases linearly with reduction in cache size.

2) Randomly Updated Warehouse Environment: We then tested a situation where obstacles were randomly added to the environment. Fig. 8 shows the planned path and the actual trajectory generated by our simulator. The random obstacles are shown in the figure as small squares. We show here only the case where the random obstacles do not completely block all passages available to the robot. From this figure, we see that some obstacles intersected the path of the robot but did not obstruct it, or were generated after the robot crossed these spots.

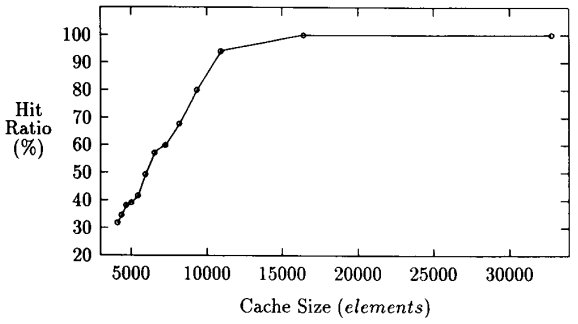


Fig. 7. Effect of cache size with aisle cache (G_3): manually updated warehouse.

From Table IV, we see that the hit ratios for the fixed cache (G_1) and the path cache (G_2) are still low. The hit count is only about half of the memory reference count. As with the results of the manually updated warehouse environment, this low hit ratio compromises the performance of the system. For the fixed cache (G_1), the total cache latency is 20.15 min. The average cache latency for each on-line global planning task is 6.7 min. For the path cache (G_2), the total cache latency is 16.9 min. The average cache latency for each on-line global planning task is 5.63 min. Since the system planned its global paths three times, the robot will spend an extra 16.9 min to reach a final destination that is reached in less than 10 min without the cache latency. The additional elements fetched in this experiment were 9866, 8858, and 2743 for the fixed, path, and aisle caches, respectively.

Fig. 9 depicts the results for the aisle cache policy (G_3). The hit ratio drops rapidly as the cache size is reduced beyond about twelve hundred elements, because the global cache can no longer hold the entire aisle data. As with the results in Fig. 7, this means that the aisle cache cannot maintain a high hit ratio unless the global cache can fetch in almost all the aisle data.

3) Randomly Updated Environment: Finally, we studied navigation in an unstructured environment where obstacles randomly appear and disappear on a floor. In this experiment, we used a random number generator to generate 250 square obstacles and then used a global path planner to plan a path, as shown in Fig. 10, where the robot has just begun to traverse its preplanned path in the low left corner. During the journey, obstacles were randomly generated and removed at the same rate (one every 3 s). Which obstacle to remove next was determined by a second random-number generator. The total number of obstacles in the environment at any time remained at 250. The final trajectory of the robot and the locations of all the obstacles on the floor at the time the robot reached its target are shown in Fig. 11.

The experimental results are listed in Table V. The results for the fixed cache (G_1) and the path cache (G_2) are a slight improvement over the results of the two previous experiments. However, the results for the aisle cache (G_3) are worse. This paradox arises because in this experiment

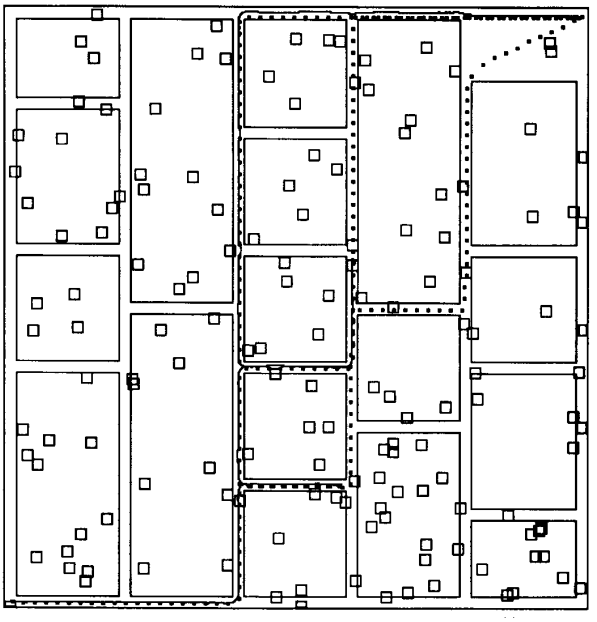


Fig. 8. Navigation with randomly updated obstacles.

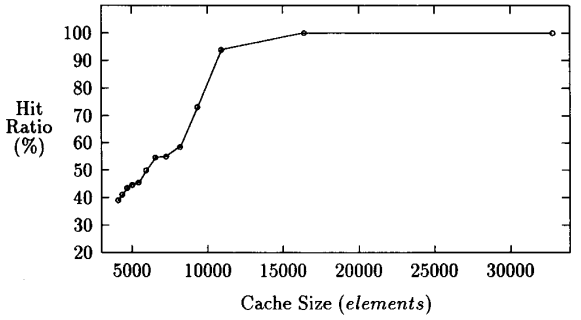


Fig. 9. Effect of cache size with aisle cache (G_3): randomly updated warehouse.

TABLE IV
GLOBAL CACHE POLICIES WITH RANDOM OBSTACLES

Policies	Hit Ratio	Hit Count	Memory Reference Count
Fixed Cache	50.98%	33 089	64 907
Path Cache	58.89%	38 226	64 907
Aisle Cache	100.00%	64 907	64 907

the global path planner was invoked only twice, as compared with three and four times in the previous experiments, so the chance of getting higher miss ratios was lowered. Conversely, the number of cache misses increase for the aisle cache since there are more newly detected obstacles in the aisles which require changes to elements in the cache. Furthermore, this time the global cache memory could not hold all the aisle data in the cache. In this experiment the additional elements fetched were 40 056, 35 820, and 17 381, respectively.

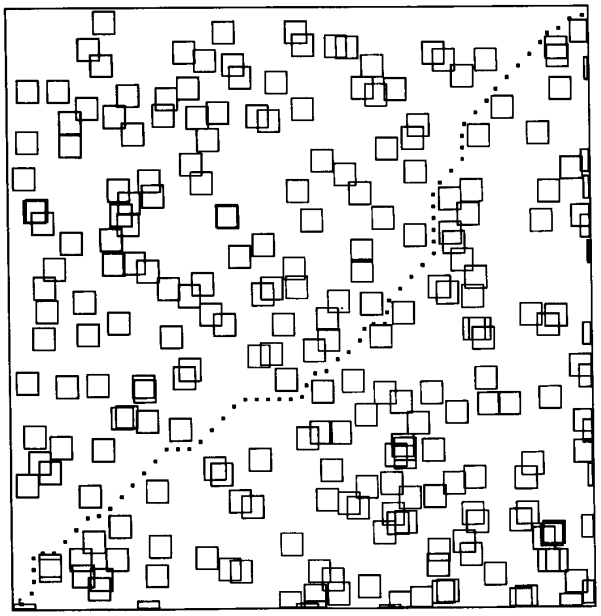


Fig. 10. Navigation with randomly added and removed obstacles: intermediate stage.

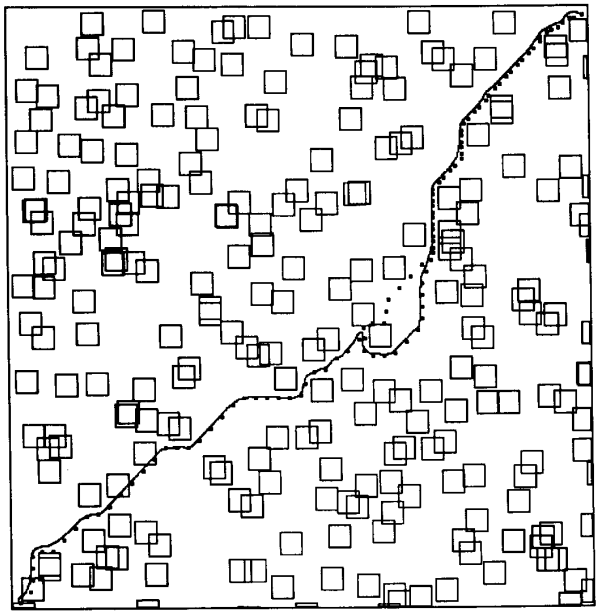


Fig. 11. Navigation with randomly added and removed obstacles: final stage.

TABLE V
GLOBAL CACHE POLICIES WITH RANDOM OBSTACLES ON A FLOOR

Policies	Hit Ratio	Hit Count	Memory Reference Count
Fixed Cache	57.38%	93 144	162 330
Path Cache	74.69%	121 239	162 330
Aisle Cache	83.43%	135 427	162 330

If we fix the local and global policies, the number of additional elements fetched indicates how dynamic the environment is. For instance, we know from the above three experiments that when a two-level cache is used for the predicted-window cache (L_2) and the aisle cache (G_3), the numbers of additional elements fetched are 66, 2743, and 17 381, respectively. We confirm from these numbers that the last environment (Figs. 10 and 11) is more dynamic than the other two (Figs. 6 and 8).

The performance of the robot system is affected by the cache latency for all three policies investigated. For the fixed cache (G_1), the total cache latency is 43.82 min. For the path cache (G_2), the total cache latency is 26.02 min. For the aisle cache (G_3), the total cache latency is 17.04 min. None of these is very promising for a robot working in this unstructured environment. These disappointing results turned our attention to the via-point cache (G_4) introduced in Section II-C.

While we did not test the via-point cache in the warehouse environment, we did test it in the randomly updated open-floor environment. In the via-point cache policy (G_4), the global path planner does not examine the full environment, but bases its planning decisions on the information contained in the planning region (Section II-C). In a warehouse environment, looking at a selected area will not guarantee that the global path planner can guide the vehicle out of trap situations. Therefore, we did not include this policy in that set of experimental studies.

Our actual implementation of the via-point cache is slightly different from our earlier description in Section II-C. The robot's position corresponds to the third via point in the cache. Thus, when the on-line global planner is invoked, it has the choice of viewing some points in the area it has just passed. This can be useful for the cases where the robot must backtrack in order to avoid a trap. Our experiments indicate that positioning the robot at the third via point is good enough. For different environments, the choice of which via point to position the robot at might vary. In our implementation, 20 via points and their surrounding elements are in the cache.

We show the results for the aisle cache (G_3) and the via-point cache (G_4) in Fig. 12 as the cache size changes. In this example, the via-point cache is much better because the aisle cache cannot guarantee locality properties for the cache. On the other hand, the via-point cache can hold enough data to ensure a very high hit ratio. Another advantage of this policy is that it does not require a large global cache. However, the success of this policy depends heavily on whether the temporary target selected during replanning generates a planning region where the planner can find a solution. This was the case in our experimental study.

C. Discussion

Several conclusions can be drawn from these experimental results. For local policies, the predicted-window cache (L_2) is better if the active-window prediction is rea-

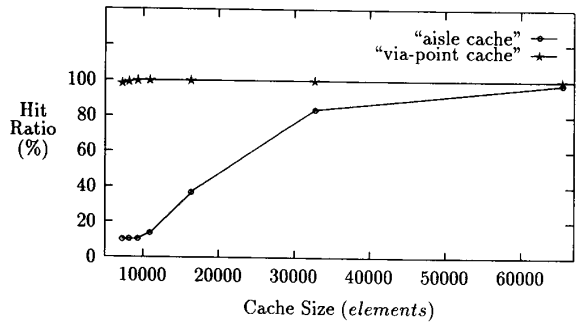


Fig. 12. Comparison between aisle cache (G_3) and via-point cache (G_4).

sonably good. For global policies, either the aisle cache (G_3) or the via-point cache (G_4) works well, depending upon the particular environment where the robots are traveling. If all the aisle data fit into the cache, the aisle-cache policy is best. If the temporary target selected for replanning generates a region where planning succeeds, the via-point cache is best. All the other policies, i.e., the one-element cache (L_3), the fixed cache (G_1), and the path cache (G_2) adversely affect robot performance. In each of these policies, the robot must pause to compensate for the long cache latency. In such circumstances, it will be better to plan global paths off-line instead. Finally, the local strategy saves more on-board memory than the global strategy, at the possible cost of losing maneuvering autonomy.

For different local and global path planners, the amount of data that can be stored in the main memory may vary. In our case, since we use a two-dimensional grid to represent local environments, only one byte is used for each element of the local data. Other implementations might be different, e.g., if three-dimensional information is needed, one byte is not enough to represent this information. Other applications may select an active window larger than the one we used if a better sensory system is available. Since the local path planner is used more frequently than the global path planner, we recommend keeping the current active window in the cache at the expense of the reducing the global cache size, if there is a cache-memory size restriction.

For single mobile-robot navigation, it may be possible to place a global path planner off-board to achieve a similar result. However, this is not the soundest strategy for multiple robot navigation in a very large environment. A centralized solution would scale very poorly in this case. Placing the global path planner in each individual vehicle will also make the system more flexible. Individual robot systems will not need to fight over the resources of a centralized planning system. A distributed approach also improves reliability. The crash of the central computer system would force all the individual robots to stop functioning.

In our implementation, the mobile robot is controlled by the local navigation routines. Therefore, the local cache policy directly affects the low-level, time-critical control routines. Let d be the deadline for a time-critical task and let t be the time the robot takes to accomplish the task assuming that all the required navigational data is in main memory. Now, $s = d - t$ is the amount of slack time available. If t_i is the time for which the ith block of data in the cache is processed before the next block of data is required, we have $t = \Sigma_i t_i$. Let us assume that a prefetching strategy is used, and that prefetching on the $(i + 1)$th block starts as soon the ith block has been fetched. If f_i is the time for prefetching the $(i + 1)$th block of data, we must have total waiting time $l = \Sigma_k (f_k - t_k)$, over all k such that $f_k - t_k > 0$. If deadlines are not to be missed, we must have $s \geq l$.

As discussed in Section II-B, the recommended cache policy in the worst case takes 76 ms (f_k) to update the active window for maneuvering activity. During one sampling period the active window needs to be updated at most once ($k = i = 1$) so that at most 30 ms (t_i) is required to process the window data. Therefore, for our experimental robot $l = \Sigma_k (f_k - t_k) \approx 76 - 30 = 46$ ms. The waiting time l of 46 ms is far below the slack time s of 470 ms where $s = d - t = 500 - 30 = 470$ ms and the deadline d for our critical task is 500 ms. Therefore, there is enough time to guarantee the execution of time-critical control routines. Furthermore, we show in that section that with our recommended cache policies, the system can even afford to miss some elements for a short period. This means that our approach has some degree of flexibility. For instance, if the deadline for a particular system is shorter than our experimental system, we can cache fewer elements than requested without missing the deadline. That is, we can afford to reduce some of cache activity. In other words, since the performance of the system is not affected much by ignoring the missed elements, this particular system can afford to do less caching (or ignore some of cache requests) in order to guarantee a deadline.

Although we only investigated one-level and two-level caches, our proposed strategies can be easily generalized to a multilevel cache. In some applications, multilevel hierarchical planning is required. The ideas presented in this paper can help to solve similar problems that arise when a multilevel planner is used with a multilayer map.

IV. Conclusions

Our interest is in the use of cache memories to deal with memory- and communication-limitation problems inherent in mobile-robot navigation in large dynamic environments, rather than in designing cache memories for high-speed computers. Therefore, we did not study cache design issues such as choice of cache size, choice of block (line) size, degree of associativity, and updating memory by write-through or copy-back approaches [5], [12]–[14].

We proposed and simulated two algorithm-implementation strategies to solve the on-board memory-limitation problem. We used a one-level local data cache for static environments and a two-level local and global cache for dynamic environments. We tested both strategies with different cache policies. To solve the bandwidth limitation

problem, we presented the predicted-window cache (L_2) and the corresponding data structures [17]. For a local cache, the predicted-window cache is the best choice. For a global cache, depending on the particular environment, either the aisle cache (G_3) or the via-point cache (G_4) could be better choice. Unlike other policies tested in simulation, none of the recommended cache policies in the suggested implementations impair the robot performance, as shown in the experiments. Our global path planner [20] searches a list of candidate nodes using a best-first strategy. One possible optimizaiton may be to cache the top k nodes on this list and their children for some suitable k. Another would be to cache the unexplored leaves of the search tree, provided the cache is large enough. We do not discuss such optimizations since we assume in this work that the low-level details of global planning are not available to the caching algorithm.

We believe that our studies and simulation provide a useful initial phase and platform for further study of single- or multiple-robot navigation. We have demonstrated that navigation in large dynamic environments can be implemented in mobile robots with the help of proper implementation strategies. We have shown how to deal with memory and bandwidth limitations, without compromising robot performance unduly, through proper choice of caching policy. Our work is likely to prove very useful to robot designers and users interested in implementing single- or multi-robot navigation algorithms in large and dynamic environments.

Appendix I
The Navigation Algorithm

The navigation algorithm used in this study is an integration of the heuristic-search and potential-field methods. The heuristic-search algorithm is used for both global path planning and trap recovery. The potential-field method is used for local path planning and path maneuver.

As shown in Fig. 13, the potential-field method [6] uses the vector sum of the virtual repulsive forces from obstacles and a virtual attractive force from a target position to provide a resultant force to guide the vehicle.

If preplanning is required, our algorithm uses heuristic search to generate a list of intermediate goals (via points) at the very beginning of the journey. If no preplanning is used, heuristic search will not be invoked at the beginning, and will be invoked only when the robot is trapped. The via points generated by the heuristic-search algorithm are used by the potential-field method to reach the final destination. For instance, whenever the robot is trapped, the heuristic-search algorithm is invoked to generate a list of via points as new temporary goals. This process continues until the robot moves to the final target specified before the journey.

The basis of our heuristic-search algorithm is a modified version of the A* algorithm invented by Hart *et al.* [4]. Our application uses a node array to represent the two-dimensional (2-D) real world as an internal map

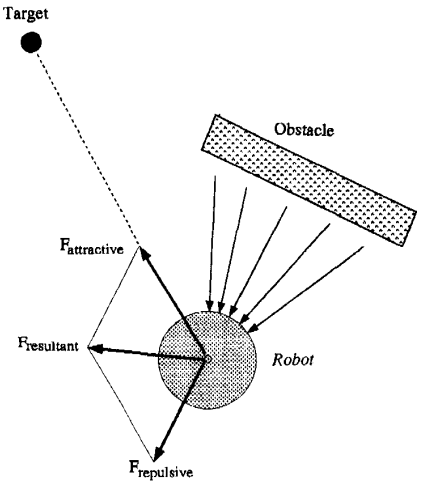


Fig. 13. Potential-field control.

stored in the memory of a mobile robot. The difference between the general A* algorithm [11] and our heuristic-search algorithm is that we include forbidden states (or nodes) in the knowledge base for each node. Forbidden states are set initially for boundary nodes, obstacle nodes, and their surrounding nodes. The forbidden states could also be determined according to safety considerations, e.g., clearance requirements. Our algorithm does not examine forbidden nodes and therefore avoids known obstacles and forbidden paths automatically. For good performance, we allow eight surrounding nodes to be generated from a current "best" node.

We have proved our navigation algorithm to be globally convergent. Convergence was also demonstrated experimentally with practical robots in both known and unknown environments. A detailed presentation and evaluation of this algorithm is described in [20].

Appendix II
Active-Window Prediction

As shown in Fig. 14, the system kinematic equations of our experimental robot are [18], [19]

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3}$$

where $0 \le v \le V_{max}$, $|\omega| \le \Omega_{max}$. V_{max} and Ω_{max} are the maximum linear and angular velocities of a particular mobile robot. Integration of (3) yields the vehicle positions

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} + \begin{bmatrix} \int_{0^+}^{t} v\cos\theta\, d\tau \\ \int_{0^+}^{t} v\sin\theta\, d\tau \\ \int_{0^+}^{t} \omega\, d\tau \end{bmatrix} \tag{4}$$

where x_0, y_0, and θ_0 are the initial positions of the vehicle.

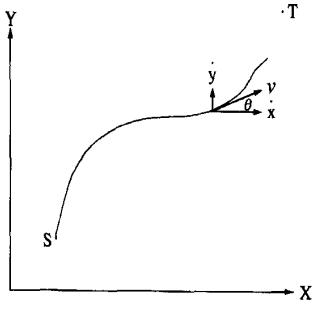


Fig. 14. Mobile-robot motion.

Considering that during very small periods v, ω, and θ are essentially constant, we obtain

$$\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} + \begin{bmatrix} v_c \hat{\gamma} T_1 \cos(\theta_c + \omega_c \hat{\gamma} T_1) \\ v_c \hat{\gamma} T_1 \sin(\theta_c + \omega_c \hat{\gamma} T_1) \end{bmatrix} \quad (5)$$

where a small hat indicates an estimated value, c denotes the current parameters, and $\hat{\gamma}$ can be estimated from (2). Equation (5) is used in our simulation for active-window prediction. This prediction is equivalent to the estimation of the next location of the vehicle.

References

[1] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots," *IEEE Trans. Syst., Man, Cybern.*, vol. 19, pp. 1179-1187, Sept./Oct. 1989.

[2] I. J. Cox, "Blanche—An experiment in guidance and navigation of an autonomous robot vehicle," *IEEE Trans. Robotics Automation*, vol. 7, pp. 193-204, Apr. 1991.

[3] E. H. Durfee and T. A. Montgomery, "Coordination as distributed search in a hierarchical behavior space," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, pp. 1363-1378, Nov./Dec. 1991.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst., Sci., Cybern.*, vol. SSC-4, pp. 100-107, July 1968.

[5] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Comput.*, vol. 38, pp. 1612-1630, Dec. 1989.

[6] O. Khatib, "Real-time obstacle avoidance for manipulator and mobile robots," in *Proc. IEEE Int. Conf. Robotics Automat.*, Mar. 1985, pp. 500-505.

[7] R. H. Katz, G. A. Gibson, and D. A. Patterson, "Disk system architectures for high performance computing," *Proc. IEEE*, vol. 77, pp. 1842-1858, Dec. 1989.

[8] J. C. Latombe, *Robot Motion Planning*. Boston/Dordrecht/London: Kluwer Academic, 1991.

[9] J. J. Leonard and H. F. Durrant-Whyte, "Mobile robot localization by tracking geometric beacons," *IEEE Trans. Robotics Automat.*, vol. 7, pp. 376-382, June 1991.

[10] C. V. Ravishankar and J. R. Goodman, "Cache implementation for multiple microprocessors," in *Proc. IEEE COMPCON*, Feb. 1983, pp. 346-350.

[11] E. Rich, *Artificial Intelligence*. New York: McGraw-Hill, 1983, ch. 3, pp. 78-84.

[12] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, pp. 472-530, Sept. 1982.

[13] —, "Bibliography and readings on CPU cache memories and related topics," *Comput. Architecture News*, vol. 14, pp. 22-42, Jan. 1986.

[14] —, "Line (block) size choice for CPU cache memories," *IEEE Trans. Comput.*, vol. C-36, pp. 1063-1075, Sept. 1987.

[15] W. C. Yen, D. W. L. Yen, and K. S. Fu, "Data coherence problem in a multicache system," *IEEE Trans. Comput.*, vol. 34, pp. 56-65, Jan. 1985.

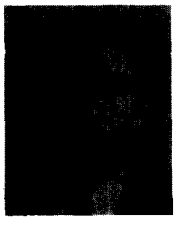
[16] Y. Zhao and T. E. Weymouth, "An adaptive route-guidance algorithm for intelligent vehicle highway systems," in *Proc. Amer. Contr. Conf.*, June 1991, pp. 2568-2573.

[17] Y. Zhao, "Theoretical and experimental studies of mobile-robot navigation," Ph.D. dissertation, Univ. Michigan, Ann Arbor, 1991.

[18] Y. Zhao and S. L. BeMent, "Kinematics, dynamics and control of wheeled mobile robots," in *Proc. IEEE Int. Conf. Robotics Automation*, May 1992, pp. 91-96.

[19] Y. Zhao and M. Reyhanoglu, "Nonlinear control of wheeled mobile robots," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots Syst.*, July 1992, pp. 1967-1973.

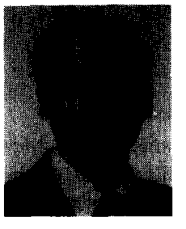
[20] Y. Zhao, "Theoretical and experimental evaluation of a local-minimum-recovery navigation algorithm," *Recent Trends in Mobile Robots*, Yuan F. Zheng, Ed., Singapore: World Scientific, 1993, pp. 75-117.



Yilin Zhao (S'89-M'92) received the B.E. degree in electrical engineering in 1982 from Dalian University of Technology, Dalian, the People's Republic of China, and the M.S.E. degree in electrical engineering in 1986 and the Ph.D. degree in Electrical Engineering Systems in 1992, both from the University of Michigan, Ann Arbor.

Before joining Motorola Inc., Northbrook, IL, as a Senior Project Engineer in 1992, he was an Instructor at Dalian University of Technology from 1982 to 1984 and a Teaching Assistant and Research Assistant at the University of Michigan from 1987 to 1991. His research interests include mobile-robot control and navigation, real-time computer systems, and Intelligent Vehicle Highway Systems.

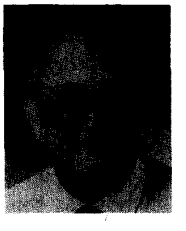
Dr. Zhao is a member of the Mobile Robots Technical Committee of the IEEE Robotics and Automation Society.



Chinya V. Ravishankar received the B.Tech. degree in chemical engineering from the Indian Institute of Technology, Bombay, in 1975, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin—Madison in 1986 and 1987, respectively.

He has been with the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor, since 1986. His teaching and research at the University of Michigan has been in the area of programming languages and distributed systems. He is a member of the Software Systems Research Laboratory and the Real-Time Computing Laboratory at the University of Michigan. His present research interests include large-scale distribution, heterogeneity, protocol synthesis, real-time systems, and database systems.

Dr. Ravishankar is a member of the IEEE Computer Society and the Association for Computing Machinery.



Spencer L. BeMent (S'61-M'67-SM'79) received the B.S.E. and M.S.E. degrees in electrical engineering and the Ph.D. degree in bioengineering from the University of Michigan, Ann Arbor.

He participated in psychophysical research in the Sensory Intelligence Laboratory from 1960 to 1967. Since then he has been a faculty member in Electrical Engineering and Computer Science at the University of Michigan, where he is a member of the Bioelectrical Science Laboratory. His research interests are in rehabilitation robotics, digital control systems, signal processing and recognition, analysis and application of solid-state electrodes, and electrical stimulation of the nervous system. He is now a Professor of Electrical Engineering and Computer Science and associated with the university's Bioengineering Program.

Dr. BeMent is a member of Sigma Xi, Tau Beta Pi, Eta Kappa Nu, and several IEEE professional groups and societies.