Mechanisms for Detecting and Handling Timing Errors

High-assurance software systems are often implemented with the dangerous assumption that timing errors will never occur.

David B. Stewart and Pradeep K. Khosla

DESIGN AND ANALYSIS OF REAL-TIME SYSTEMS IS HEAVily based on knowing worst-case execution times (WCET) of periodic threads and aperiodic servers. Accurately measuring WCET, however, is often difficult and sometimes impossible, for several reasons:

- Interrupts in the system, which either execute longer than expected or occur more frequently than anticipated, may steal critical execution time from the highest priority threads.
- Variations in processing speed due to caching, pipelining, and bus arbitration may alter WCET.
- There is no easy way to accurately measure execution times of embedded code.

As long as scheduling policies are based on WCET, these difficulties in measuring WCET inevitably lead to timing errors in the system. Many of these errors go undetected until more catastrophic failures occur, and others result in the system failing to meet its specifications, but with non-obvious reasons for the cause of such failures.

We have created low-overhead policy-independent real-time operating system (RTOS) mechanisms, which detect and handle these types of timing errors. The mechanisms can be used with a variety of common scheduling algorithms, and serve as the basis for easily extending these policies to incorporate aperiodic servers, soft real-time threads, imprecise computations, and adaptive real-time scheduling. The mechanisms have been incorporated into the Chimera RTOS [9].

There has been extensive research into detecting and handling hardware, software, and state errors. A hardware error is a result of failing hardware or a processor-generated exception. A software error is a mistake either in the design or implementation of software. A state error is a discrepancy between the current state of the system, and the internal data representation of that state.

However, there has been little research into detecting and handling timing errors. These types of errors, which occur in real-time systems, are failures to meet the timing specifications of the system. Much research in real-time systems focuses on developing scheduling policies that can be used to guarantee that timing errors will not occur.

Unfortunately, these hard real-time systems are often implemented *with the dangerous assumption that due to correct design, analysis, and testing, a timing error will never occur.* Since many of these policies are based on estimates of the WCET of each thread, the reliability of the system is often a function of the accuracy of the estimates. These systems rely on the guarantees of the scheduling policies to meet the timing requirements, and cannot detect missed deadlines, let alone take emergency action in response.

Yet errors in these systems do occur, primarily due to the difficulties in accurately measuring the WCET of the code. The most common timing error is the *missed deadline*, where a real-time thread fails to complete its execution on time.

Another timing error, which is perhaps just as common but not necessarily as obvious, is the *incorrect estimate of execution time*. The guarantees provided by a scheduler are only valid if the WCET times used in the scheduling analysis are accurate, or at worst, over-estimated. What if those estimates are wrong? Executing a real-time thread for longer than originally anticipated usually leads to unexpected overloading of a CPU. Consequently, even threads that were guaranteed to meet all deadlines eventually miss one!

To our knowledge, there has not been any prior solutions for policy-independent mechanisms at the operating system level that can be used to detect and handle timing errors.

Several real-time programming languages, such as Real-Time Euclid [4] and RTC++ [2], describe policy-independent mechanisms for timing errors. Although the handling mechanisms are part of the language, they still depend on a real-time kernel to detect and notify the language of the timing errors. They also require that all code be written in that language. The operating-system based mechanism we describe is compatible with any language, and can be used as the basis for notifying languages requiring kernel support.

```
float Ta = 0.1;/* period of thread, in seconds */
float Ea = 0.04/* estimated WCET of thread (seconds) */
float nextstart;/* start of next cycle */

main(args) {
    /* initialization stuff goes here */
  nextstart = clock(); /* store 'now' into nextstart */
  while (1) { /* begin periodic loop */
    nextstart += Ta;/" compute next restart time */
    pause(nextstart,Ea,Ta)
    {execute one cycle of thread here;}
  }
}
```

Figure 1. Framework for periodic threads in Chimera

Flex [3] is a language extension to C++ that includes built-in timing error detection and handling for imprecise computations, without relying on the operating system kernel, as do the other language-based methods. The mechanism uses a combination of delay calls and alarm functions to implement the detection and handling. This mechanism, however, is tied to the imprecise computation model [7], and, as stated by the authors, there is significant overhead that slows down typical C++ code 2 to 10 times.

Other efforts to identify timing errors have been limited to using monitoring methods, which are typically only available during the development phase of a real-time system.

Detection and Handling of Timing Errors

The programmer specifies the deadline and maximum execution time whenever a thread waits until the start time of its next cycle arrives. Chimera's C language framework for a periodic thread is shown in Figure 1.

The system call that interfaces with the Chimera microkernel is *pause(float restart, float exectime, float deadline)*. It programs one of Chimera's virtual timers to wake up the thread at *restart* time. Time is specified in seconds. When the thread begins executing its next cycle, it will be allowed to use a maximum of *exectime* seconds of the CPU. It must also complete its execution before *deadline*, which is specified in seconds relative to the start of the thread.

In the example in Figure 1, the maximum execution time E_a is 40msec (note: msec=millisecond, μsec=microsecond), and the deadline in this case is the start of the next cycle, thus the deadline is set to the thread's period T_a. The deadline can also be earlier than the start of the next period, such as when using the deadline monotonic algorithm [1]. Although in the example the period and execution time are hard-coded, it is more typical to pass these values as arguments to a thread. In addition, the initial static priority of the thread is set when the thread is created by a parent thread. However the static priority can be temporarily modified for failure handling and execution of aperiodic servers, as discussed later in this article.

The timing constraints in Chimera are specified within a thread using the policy-independent *pause()* interface, so that it can be used with a variety of static and dynamic scheduling algorithms.

Detection Mechanism

Timing errors are detected jointly by the scheduler and the microkernel. The implementation of the timing error detection in the Chimera RTOS is described in this section.

The scheduler helps to track missed deadlines by identifying the earliest deadline among all threads on the READY and BLOCKED queues. The running thread is on the READY queue. The BLOCKED queue is for threads waiting for a semaphore or for a message. Threads awaiting their next restart time are on a separate PAUSE queue and do not yet have deadlines.

The scheduler programs the earliest deadline into one of the microkernel's virtual timers, used to monitor missed deadlines. If the current time reaches this programmed

```
float Ea = 0.05; /* estimated execution time, in seconds */
float Ta = 0.1, nextstart;
int Ph=100; /* highest priority in system */
extern MSG *msgq; /* message queue for IPC */
jmp_buf restart;

tfhandler(int type) {
    switch (type){
    case DEADLINE:      /* missed deadline */
        msgSend(msgq,"no more time,no data this cycle");
        longjmp(restart);
        return;
    case MAXEXEC:       /* effective CPU time used up */
        shutdown();     /* emergency shutdown;doesn't return */
        }/* end switch */
}
main
    { initialization stuff goes here }
       /* execute handler with highest priority */
    tfhInstall(tfhandler,Ph);
    nextstart = clock();
    while (1) {         /* begin periodic loop */
       setjmp(restart);
       nextstart += Ta;
       pause(nextstart,Ea,Ta);
       { execute one cycle of thread here; }
    }
}
```

Figure 2. Example of a timing failure handler (TFH) in Chimera

time, the microkernel calls the policy defined in the scheduler, using the missed deadline entry point, and determines which thread(s) missed a deadline. It then invokes a handler for those failing thread(s).

In order to control maximum execution time, the microkernel maintains a software down-counter. When a thread begins to execute, the maximum execution time it requires is copied into this counter. On every clock tick, the counter is decremented. If it reaches zero, then the thread has used up its allotted CPU time, and a timing error has occurred.

If a thread blocks or is preempted during execution, then the value in the down-counter is saved as part of the context of the thread. When the thread is swapped in again at a later time, the down-counter value is restored to its saved value.

A thread's cycle is considered complete when it calls the *pause()* routine again. At this time a new deadline for that thread is specified, and the maximum execution time for the thread's next cycle is replenished. If the thread's old deadline was programmed into the virtual timer, then the scheduler finds the next earliest deadline and reprograms the timer. Otherwise, the timer continues to run, as the thread with the earliest deadline is not the executing thread (this situation should not occur if the EDF algorithm is used).

Handling Mechanism

When a timing error is detected, a user-defined *timing failure handler* (TFH) is invoked. The TFH executes within the context of the failing thread. Failure handlers can perform any kind of recovery operations, such as aborting the thread and preparing it to restart the next period, continuing the thread and forcing it to skip its next period, sending a message to some other part of a distributed system, performing emergency handling such as a graceful shutdown of the system or sounding an alarm, or, in the case of iterative algorithms, returning the current approximate value regardless of precision.

Since a TFH can be called at any time, it must be designed as *re-entrant* code, similar to that used for interrupt and signal handling. It saves part of the thread's context (such as scratch registers), so that the code can be called at any time, without compromising the integrity of the main body code.

High-level languages such as C do not provide the necessary functions for creating re-entrant code. As a result, this code must be written in assembly language, and it is usually not portable. In Chimera, a generic assembly language timing error handler was created. This code performs a jump-table lookup, and calls a C-language subroutine corresponding to the thread that has the timing error.

The user creates a TFH by defining a subroutine, then installing it using the *tfhInstall(funcptr handler,int hpriority)* routine. The *handler* argument is the name of the subroutine, while the *hpriority* is the static priority at which the failure handler is to be executed.

The *hpriority* parameter informs the microkernel to temporarily modify the priority of the thread during execution of the handler. This feature allows failure handling code to have a priority different from the failing thread. Thus critical failure handling can have high priority and can be called immediately, while failure handlers for soft real-time threads have lower priority, and do not use execution time of other more critical threads. If *hpriority* is 0, then the handler is executed with the same priority as the failing thread.

An example of defining and installing a TFH is shown in Figure 2. The argument passed to the failure handler is the type of failure which is either DEADLINE or MAXEXEC. This allows each thread to have separate handling for the *missed deadline* and *maximum execution time used up* timing errors respectively.

In this example, the TFH is designed to send a message to another thread and prepare the thread to restart execution if a missed deadline occurs, and gracefully shuts down the system if the thread uses more than its allotted WCET.

```
float Cmds = 0.05;     /* in seconds */
float Tmds = 0.1, nextstart;
int Ph=100,Pl=50; /* assume threads in critical set */
                  /* have priority between Ph and Pl. */

dshandler(int type)
{
   switch (type){
   case DEADLINE: /* type 1 failure; replenish server */
     set_priority(Ph);
     nextstart += Tds;
     set_deadline(nextstart,Cds,Tds);
     return;

   case MAXEXEC:  /* typs 2 failure; capacity used up */
     set_priority(Pl);
     return;
   } /* end switch */
}
thread_name()
{
    { initialization stuff goes here }
       /* execute handler with default priority */
     tfhInstall(dshandler,0);
     nextstart = clock();
     set_deadline(nextstart,0,Cmds,Tmds);

     while (1) {       /* begin periodic loop */
         msgReceive(message);  /* wait for event */
         { do event handling here; }
     }
}
```

Figure 3. Framework for a deferable server using a TFH to control the server's execution

An issue in designing the mechanisms for supporting TFHs is to do so within the scope of the failing thread. Interrupt handlers execute within the scope of the kernel, and not that of any thread. As a result, interrupt handlers do not have access to most of a thread's data. Also, the failing thread may not necessarily be the thread that is currently running. Particularly for the missed deadline timing error, it is likely that the thread is not executing, but rather is on the READY or BLOCKED queue.

In our Chimera implementation, the microkernel modifies the stack and program counter of the stored context. The current program counter is added to the stack, and the stack pointer adjusted accordingly. The program counter is then modified to contain the start address of the generic TFH re-entrant code. As a result, the next instruction executed by the thread will be the TFH. The priority of the thread is also modified to *hpriority*, and a re-schedule is performed if necessary. If the failing thread now has the highest priority in the system, it will be the next thread selected. Otherwise, the TFH will be executed only when it is that thread's turn to execute, based on the real-time scheduling policy in use at the time.

Modifying a thread's stored context works for all cases except when the running thread fails. In this case, the running thread is swapped out first, then the modifications are made. If the thread still has highest priority it is swapped back in, otherwise, the next highest priority thread is selected.

After executing a TFH, there are three possibilities to resume processing: restart, continue, or exit. The restart option causes the thread to return to the beginning of its code, using a Unix-like *setjmp()/longjmp()* mechanism. The *continue* option allows the thread to keep executing from the time before its code was interrupted by the TFH. This is accomplished by simply doing a *return()*, since the old PC was saved when the thread's stored context was modified. The *exit* option is used to kill the thread altogether.

Since a failure handler can be called at any time, it might be called when the failing thread is writing data in a critical section. In such cases, execution of the TFH must be delayed until the end of the critical section to ensure the integrity of the data. This is accomplished by locking out the TFH during such critical sections. The thread continues to execute at the same priority to the end of the critical section, unless the failure handler is defined to have a higher handler priority, in which case the thread immediately inherits the higher priority. This in effect extends the definition of the TFH to implicitly include a "finish writing data first." Chimera provides a facility for locking out a TFH, or such code can be called by an RTOSs semaphore or condition variable mechanism.

If a thread is on the BLOCKED queue, it is moved to the READY queue, with the next instruction to execute being the TFH. In this case, if the *continue* option for the thread is selected, the thread returns to the BLOCKED queue.

The latency to call a TFH is the length of one context switch, plus 12 µsec (on a 25MHz MC68030) for updating the stored context and executing the generic assembly language framework. This overhead is incurred *only* when a timing error occurs. The overhead of decrementing the down-counter on each clock tick and checking if a deadline has been missed is less than one microsecond.

Timing Error Handling Policies

Given mechanisms for timing error detection and handling, it is reasonable to ask what is the best way to use them. In many cases, designing a system to expect timing errors, then handling them appropriately, can result in more straightforward scheduling policy implementation.

In this regard, we discuss how the mechanisms can be used for more advanced scheduling. This includes scheduling aperiodic servers, soft real-time threads, imprecise computations, and adaptive systems. The mechanisms are compatible with popular real-time scheduling policies, such as the rate monotonic (RM) [6] and deadline monotonic [1] static priority algorithms; and the maximum-urgency-first (MUF) [10], and earliest-deadline-first [6] dynamic priority algorithms.

Aperiodic Servers

Aperiodic servers are designed to handle events that arrive at random times in a hard real-time system. Initial implementation of two types of aperiodic servers, the deferable server [5] and sporadic server [8] into the Real-Time Mach operating system [12] required that the scheduler be modified to explicitly support them.

In this section, we show how the timing error detection and handling mechanisms can be strategically used to control the capacity and execution of aperiodic servers. As a result, any system that supports these timing error mechanisms can also support the aperiodic servers *without the need to modify any part of the real-time scheduler.*

The aperiodic servers were initially designed for the RM algorithm, and later extended for use with the MUF algorithm. The implementation of these aperiodic servers is basically the same; only the scheduling analysis differs [10].

Deferable Server: The basic concept behind the deferable server (DS) is that a periodic thread with highest priority is created with a fixed period T_{ds} and maximum capacity C_{ds}. The *capacity* represents the maximum CPU time that the server can use within one period. If more than that amount of CPU time is required, then the server's priority is lowered, until the beginning of the next cycle [5].

The DS executes with the highest priority of P_h. In comparison, all the periodic threads which form the critical set would have a priority less than P_h. T_{ds} is equal to that of the highest frequency periodic thread that may execute. Its capacity C_{ds} is equal to U_{ds}/T_{ds}, where U_{ds} is the DS size. The server size represents the maximum execution time per cycle that the server can use without jeopardizing the execution of any of the periodic threads in the critical set. The server size can be analytically derived, as described in [8] when using the RM algorithm, and [10] when using the MUF algorithm.

The structure of the main body of a DS is similar to that of a periodic thread, and is shown in Figure 3. The maximum execution time of the thread is set to the capacity, C_{ds}, and the deadline is set to the thread's period T_{ds}. The routine *dshandler()* is the TFH that is called when either the server's capacity has expired or the server's replenishment time has arrived.

If a timing error occurs as a result of the thread using up all its allotted CPU time (i.e., the error is of type MAXEXEC), then the server has used up its capacity, and must let the other critical real-time threads execute so that they do not miss their deadlines. As a result of using up its time, the microkernel signals a timing error, and the TFH for the DS is called. The TFH lowers the priority of the thread to P_l, where P_l is a lower priority than any periodic threads in the critical set, but still greater than threads that are not in the critical set. The server continues to execute only if there is leftover CPU time after the critical set threads have executed, or if its replenishment time arrives.

The replenishment time of the server is its deadline. When the thread's deadline time arrives, the microkernel detects this as a timing error, and again calls the TFH. This time, however, the error is of type DEADLINE, which implies a missed deadline. For the server, however, this is an indication that its capacity can be replenished. Therefore, the TFH resets the criticality of the server back to P_h, and sets up new deadline and resets the maximum execution time back to C_{ds}. Note that the routine *set_deadline()* is similar to *pause()* in terms of setting the deadline and maximum execution time, but it leaves the thread on the ready queue, rather than pausing the thread until the start time arrives.

If using the MUF scheduling algorithm instead of RM, the only difference is that the criticality of a thread is adjusted rather than the static priority [10].

Sporadic Server: The sporadic server (SS) is similar to the DS, except that the method in which the thread's capacity is replenished is different. In order to improve on

```
float Css = 0.05;      /* in seconds */
float Tss = 0.1, nextstart;
int Ph=100; /*highest priority thread in critical set*/
int Pl=50;  /*lowest priority thread in critical set*/

struct qlist {
    float t,c;
    struct qlist *next
} Q[QMAX];
    /* Q[0] is header node */
qList *qHead=&Q[0], *qTail=&Q[0],*qFree=&Q[0];

sshandler(int type) {
      /* both DEADLINE and MAXEXEC types of deadlines */
      /* handled same way */
   mexec = execleft();
   while (q->t <= clock() {
      mexec += q->c;
         /* delete entry from list */
      qHead = q->next; q->next = qFree; qFree = q;
   }
   if (mexec == 0) {
         /* no exec time left, go to lower criticality */
      set_deadline(clock(),Infinity,q->t-clock());
      set_priority(Pl);
   }else {
        /* some high priority exec time left */
      set_deadline(clock(),mexec,Infinity);
      set_priority(Ph);
   }
}
thread_name()
   float estart,eend;
   { thread-dependent initialization stuff goes here }
   { initialization of qList goes here }
       /* handler should execute with default priority */
   tfhInstall(sshandler,0);
   nextstart = clock();
   set_deadline(nextstart,Css,Infinity);

   while (1) {      /* begin periodic loop */
       msgReceive(message);  /* wait for event */
       estart = execstart();
       { do event handling here;}
       eend = execend();
          /* add entry to list */
       q=qFree; qFree=q->next;qTail->next=q;
       q->t=estart+Tss; q->c=eend-estart;
    }
}
```

Figure 4. Framework for a high-priority sporadic server using a TFH to control the server's execution

CPU utilization and server size, the SS replenishes the capacity only when necessary, and sometimes it is only partially replenished. In contrast, the DS always replenishes its capacity fully at its deadline time. This change results in a more complex aperiodic server, but the gain in CPU utilization and server size might be worth the additional complexity [8].

The design of an SS is similar to that of the DS, except that instead of merely specifying a server's capacity and its replenishment time, a list of execution start times and the amount of execution used must be maintained. The framework for implementing an SS in Chimera is shown in Figure 4.

Initially, the maximum execution time of the SS is C_{ss} and its deadline time is set to infinity. This means that the TFH will only be called when the maximum execution time reaches zero, and not as a result of any kind of missed deadline. Replenishment occurs whenever the server exhausts its execution time. The amount of replenishment depends on when the CPU time was used up, which is maintained in a list.

Whenever an event is processed, the thread updates the replenishment time of the server based on when the event's processing began and how much execution time was used. The routines *execstart(), execend(),* and *execleft()* are calls that return information about the execution time being used by that thread; the microkernel stores these values in the control block of each thread as part of the timing error detection mechanism.

Soft Real-Time Threads

An obvious use of the timing error detection and handling mechanism is to implement soft real-time threads. For example, one scheduling policy may be for a thread that misses deadlines to continue to execute, but to use up the time from its next cycle as well. This in effect temporarily halves the frequency of that one thread. This method works well especially for threads which receive input from analog sensors, where missing an occasional sensor reading is acceptable. The schedulability analysis of a system containing this type of soft real-time thread is given in [10].

Implementation of these threads in Chimera is the same as for the hard real-time thread shown in Figure 1, except that it uses a TFH similar to the one shown in Figure 3 for the DS.

When a MAXEXEC error occurs, the priority of the thread is lowered such that it is less than the priority of any thread in the critical set. For the remainder of that cycle, it only executes with low priority. When the thread's deadline time arrives (i.e., a DEADLINE error occurs), the thread's priority is restored, and the thread can continue to execute with a priority that guarantees some execution.

Imprecise Computations

Another possible use of our mechanisms is for implementing imprecise computations. This policy provides the means to create approximate computations quickly, and to only fully execute a process if time permits.

A thread that can be scheduled using an imprecise computation model typically has two parts, an essential part and an optional part. If a missed deadline occurs while the optional part is being executed, the optional computations are discarded and only the results of the essential part are used. Scheduling theory using an imprecise computation model is summarized in a collection of papers in [7].

Adaptive Real-Time Scheduling

In dynamically changing systems, it may not be possible to analyze the schedulability of a thread set a priori. Streich developed a policy for adaptive scheduling of soft real-time threads, called *TaskPair Scheduling* [11]. It is based on the notion of optimistic case execution times, in addition to WCET.

The method requires that a built-in monitor constantly measures execution times of threads, and that a timing error handler will be invoked if the optimistic execution time is used up.

The mechanisms described in this article, in conjunction with Chimera's automatic task profiling [10], provide the necessary functionality to implement adaptive real-time scheduling policies such as TaskPair scheduling.

Summary

We have designed policy-independent mechanisms for detecting and handling timing errors. The mechanisms requires less than 1 μsec overhead per reschedule operation, and have a latency approximately the length of one context switch for handling an error. We are investigating techniques for integrating timing error detection and handling with other non-timing error handling techniques.

References

1. Audsley, N.C., Burns, A., Richardson, M.F., Wellings, A.J. Deadline monotonic scheduling theory. In *Proceedings of IFAC Workshop on Real Time Programming (WRTP '92)* (Bruges, Belgium, June 1992), pp. 55–60.
2. Ishikawa, Y., Tokuda, H., and Mercer, C. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of ECOOP/OOPSLA* (Ottawa, Canada, Oct. 1990), 21–25.
3. Kenny, K. and Lin, K-J. Building flexible real-time systems using the flex language. *IEEE Computer 24*, 5 (May 1991), 70–78.
4. Kligerman, E. and Stoyenko, A.D. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. Softw. Eng. SE-12*, 9 (Sept. 1986), 941–949.
5. Lehoczky, J. Sha, L. and Strosnider, J.K. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium* (Dec. 1987), pp. 261–270.
6. Liu, C.L. and Layland, J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM 20*, 1 (Jan. 1973), 44–61.
7. Natarajan, S., Ed. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, Boston, 1995.
8. Sprunt, B., Sha, L. and Lehoczky, J. A periodic task scheduling for hard real-time systems. *J. Real-Time Systems 1*, 1 (Nov. 1989), 27–60.
9. Stewart, D.B., Schmitz, D.E. and Khosla, P.K. The Chimera II real-time operating system for advanced sensor-based control applications. *IEEE Trans. on Systems, Man, and Cybernetics 22*, 6 (Nov./Dec. 1992), 1282–1295.
10. Stewart, D.B. Real-time software design and analysis of reconfigurable multi-sensor based systems. Ph.D. dissertation, Carnegie Mellon University, April 1994.
11. Streich, H. TaskPair scheduling: An approach for dynamic real-time systems. *Intl. J. of Mini and Microcomputers 17*, 2 (1995), 77–83.
12. Tokuda, H., Nakajima, T. and Rao, P. Real-time Mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop*, (Oct. 1990), 73–82.

DAVID B. STEWART *(dstewart@eng.umd.edu) is an assistant professor of Electrical Engineering and Principal Investigator in the Institute for Advanced Computer Studies, University of Maryland, College Park. For more information, see http://www.ee.umd.edu/~dstewart.*

PRADEEP K. KHOSLA *(pkk@cmu.edu) is a professor of Electrical and Computer Engineering and Principal Investigator in the Robotics Institute, Carnegie Mellon University.*

Research presented in this article was primarily performed as part of the Chimera Project. Chimera has been funded in part by NASA, ARPA, Sandia National Laboratories, and the Electrical and Computer Engineering Department and the Robotics Institute at Carnegie Mellon University. Continuing research on this topic is supported by the Electrical Engineering Department and the Institute for Advanced Computer Studies at University of Maryland, College Park.