Exploiting Computational Parallelism with a Hierarchical Robot Control System

John L. Michaloski, Thomas E. Wheatley and Ronald Lumia
National Bureau of Standards
April 1988

Abstract

Sequential machines do not provide enough computational performance for a real-time robot control system. Hierarchical structuring will be considered not only as a software development methodology, but also as a means for implementing a robot control system on a pipelined parallel architecture to maximize performance. The concept of "virtual control loops" will be introduced as a framework in which to discuss execution and communication of parallel levels within a hierarchical control system. Timing requirements of a hierarchical control system and control level will be modeled. Timing analysis will motivate the discussion for control levels that exhibit a short-term executor monitoring cyclic response behavior and a long-term planner anticipating the future. A control level software template will be presented that combines concurrent executors and planners with interprocessor communication.

Keywords

real-time, hierarchical control, task decomposition, virtual control loops, response time, fine granularity, coarse granularity, timing, communication

1.0 Introduction

The use of multiple processors is beneficial for a number of reasons, including better cost performance, modular growth, increased reliability through replication, and flexibility for testing alternate control strategies via different partitioning [8, 16]. However, reaping the

---

The authors current address is the National Bureau of Standards, Robotics Systems Division, Bldg. 220 Rm. B124, Gaithersburg, MD. 20899

benefits of parallel processing does not result from simply adding more processors to a system. The effectiveness of a parallel implementation depends on the inherent parallelism of the algorithms and the overhead of interprocessor communication. Algorithms that are purely sequential will not run faster on a parallel machine. Further, the cost of interprocessor communication is a larger factor than it would be on a sequential machine using procedure calls. Interprocessor communication using shared memory or message passing takes longer because of the overhead of protocols and extra synchronization. To achieve the benefits of parallelism, the cost of communications between processors must not exceed the time savings obtained by parallel execution on the different processors.

Allocating multiple processors to algorithms is usually defined in the context of a fine-grain versus coarse-grain approach [12, 13, 14]. An entire process which is allocated one processor for the life of its execution could be considered coarse grain. Computational-intensive problems (such as the dynamics of a robot arm, specifically a matrix multiply in parallel) or data-intensive problems (vision processing) utilizing numerous processors in parallel to compute solutions are both examples of a fine grained processing approach.

Parallel machines exhibit different system capabilities. General-purpose coarse-grain machines are a type of parallel computer that maximize batch throughput. In this case, the user may be unaware of any parallelism involved. Supercomputers are coarse-grained machines that specialize in fine-grained number crunching in parallel for mathematically intensive algorithms. In this case, the programmer may add some data flow enhancements to assist the number crunching. Other fine-grain computers specialize in data-intensive operations that do not handle general-computing efficiently. The variety in machines leads to the distinction between systems that maximize the throughput of many jobs, known as *throughput-oriented multiprocessors*, and systems that maximize the execution of one process, known as *speedup-oriented multiprocessors* [5].

This paper will focus on those parallel architectures that best fit the system requirements of an intelligent robot controller in terms of price versus performance. A robot controller is a large system that is composed of layers of fine-to-medium-grained processes. A robot controller can be characterized as a speedup-oriented multiprocessing application

since the controller is partitioned into a set of concurrent, cooperating processes. A multiprocessor system sharing a common or enhanced bus is an example of an architecture that offers a mix of capabilities that can accommodate this architecture diversity. A goal of an efficient multiprocessor system is to exploit the benefits of parallelism while minimizing the impact of parallelism on the software algorithms. Hierarchical structuring of such a control system offers an easy and systematic parallel approach. With a hierarchical control system, levels in the hierarchy can be developed as sequential processes and then parallel integration can be modeled as communicating sequential processes [10]. Thus, a control system can be developed functionally independent of the implementation and then hierarchically integrated with the communicating sequential processes model.

2.0 Hierarchical Real-Time Control

Hierarchical decomposition for a control system is a well-defined structuring technique that can be used for organizing the design so that channels of timing, communication and authority are well-established [1, 4, 15]. As implied by the name, a hierarchical system is comprised of many levels working together on a common goal [19]. When applied to control processes, the hierarchical structure is not to be confused with a deeply nested serial process that follows a single thread of control flow. Instead, when composed as a system, hierarchical control is built as layers of *virtual control loops*. When executing, each virtual control layer in the hierarchy can be considered as part of a long chain defining the hierarchical state, yet each level's action is based on its own control flow. Much as a computer executes an instruction within a given duty cycle, the virtual control loops correspond to layers of software modeled analogously to a physical machine. The virtual control loop software exhibits cyclic feedback behavior that samples inputs including command and status and guarantees some output within a given time.

Each virtual control layer communicates to other layers through well-defined command and status interfaces. Levels within the hierarchy can be considered independent so that modification to one layer to improve performance does not affect the operation of other layers and can lead to standardized classifications. The independence of control flow between layers allows virtual control loops to be easily ported onto a pipeline parallel computer architecture. Parallelism is not a direct result from hierarchical decomposition into virtual control loops levels in the control system, because a lower level cannot process without a

command from its adjoining higher level. However, once a task is underway a pipeline is created such that each level is executing in parallel at a different stage of completion towards the goal. For example, it would take 5 control cycles for an emergency abort from the top level to filter down to the bottom level in a five layer hierarchy. This leads to the problem of handling information in a pipeline. Each level communicates every cycle, so that it would take i cycles to move data up or down i levels in the hierarchy. Since each level is at various stages of completion of a task, coordinating timing information between levels must be acknowledged.

From experience, an implementation of a hierarchical control system based on these parallel pipelined concepts is both robust and effective because of the structure imposed on the software. The use of levels offers the benefit of information hiding, so that software design and development can concentrate on local problems instead of attempting to solve problems globally. Further, the addition of a system-wide synchronization pulse where the levels execute in lock step adds the dimension of comprehensibility to the system. In general, parallel systems are difficult to understand. With a system heartbeat, execution can be characterized as a state machine where transitions are predictable and repeatable. This does not imply that all software is rigidly defined with a state transition mechanism. Rather, software abstraction is adjustable with selective degrees of resolution ; much like changing the magnification of a microscope. At the highest level of abstraction, the state transition are defined as the commands and status exchanged. This allow easy pinpointing of problems within the hierarchy. Tracking execution with a finer resolution of abstraction relies on the basic state transitions employed by the computer. From a software development standpoint, testing and analysis is much easier with a system that allows a selectable resolution of software abstraction.

Two design factors are at the heart of exploiting the benefits of a hierarchical control system. The notions of task decomposition and response time are crucial in determining the number of processors in a hierarchical system, the complexity of any level, and the communication.

2.1 Task Decomposition

Task decomposition can be defined as the process of recursively breaking down a task into smaller more manageable tasks, until some atomic level of activity is reached. At each level in the hierarchical breakdown, an interface exists where the adjoining levels exchange information. The higher level communicates a command to its neighboring lower level. Likewise, the lower level command communicates a status to its neighboring upper level. This communication protocol is analogous in sequential programming to a subroutine invocation as a command and a return value as a status. Figure 1 illustrates the data flow between levels.

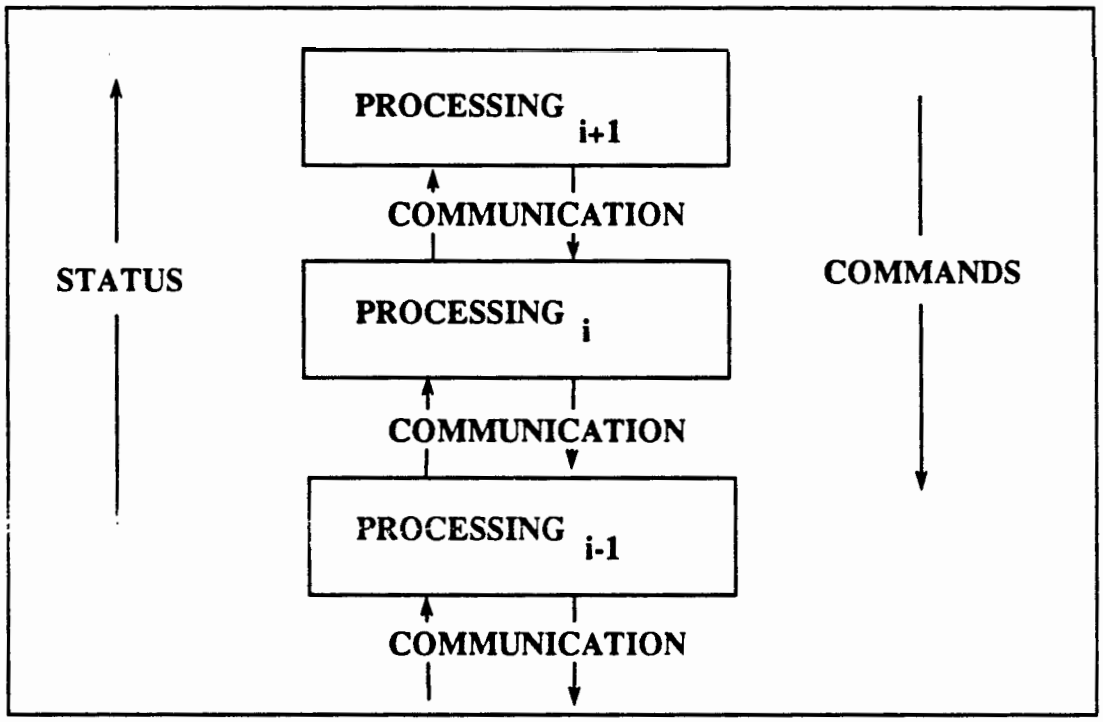

Figure 1. Information Flow in a Hierarchical Control System

Defining a hierarchal decomposition for a control system of a robot starts by taking a high level task and through a series of task decompositions reduce this task to a set of motion primitives and at the same time status of the environment would filter up the hierarchy. Although task decomposition breaks each task down into smaller and smaller tasks, each lower level is not completely dependent on its neighboring upper level. Although

levels may share some data that models the world, each level can be considered to run independently of each other, responding to a command, and supplying a status much like a plant in a normal feedback control system. Thus, each level can be considered a virtual control loop.

The amount of decomposition at any level should be based on reducing the problem to a set of well-defined, manageable sub-tasks. If any of these sub-tasks should become too complex, then another level should be added to the hierarchy and a new decomposition attempted. By breaking down a series of tasks, duplicate subtasks should exist across task definitions. The final collection of unique sub-tasks forms the basis of the lower level commands. With a robust collection of lower level commands, future tasks can be decomposed and defined as a series of these existing lower-level commands. This feature bounds the software development so that enhancement on one level should not require modifications to a lower level. Thus with a well-defined and complete set of commands, the control system is extensible and adaptable to handle new functional needs without requiring major reconstruction of the software. Figure 2 below illustrates how a robust set of subtask definitions are created while defining a set of tasks, and how extensibility for future task definitions results as a by-product.

$T_1 = \{ ST_a, ST_b, ST_d, \}$

$T_2 = \{ ST_c, ST_b, ST_d, ST_a, ST_f \}$

$T_3 = \{ ST_b, ST_a, ST_c \}$

Set of Subtasks = $\{ ST_a, ST_b, ST_c, ST_d, ST_e, ST_f \}$

Define New Task : $T_4 = \{ ST_b, ST_d \}$

Legend

T_i = Task definition i

ST_x = SubTask Definition x

Note : ST_e is unused but available for future use

Figure 2. Task Decomposition

Decomposing a single task consists of creating a list of events that will occur in the

process of completing the task. The occurrences of events depend on the system state and result in actions in the system. Actions correspond to the process or system output that provides the appropriate control to achieve the desired goal. A description of the state of the machine can be done as a set of prerequisite conditions. An event occurs when a vector of prerequisite conditions are true. An event at one level is treated as a new precondition status at the higher level and results in a new commanded goal to the lower level. Figure 3 illustrates the composition of subtasks from a set of preconditions.

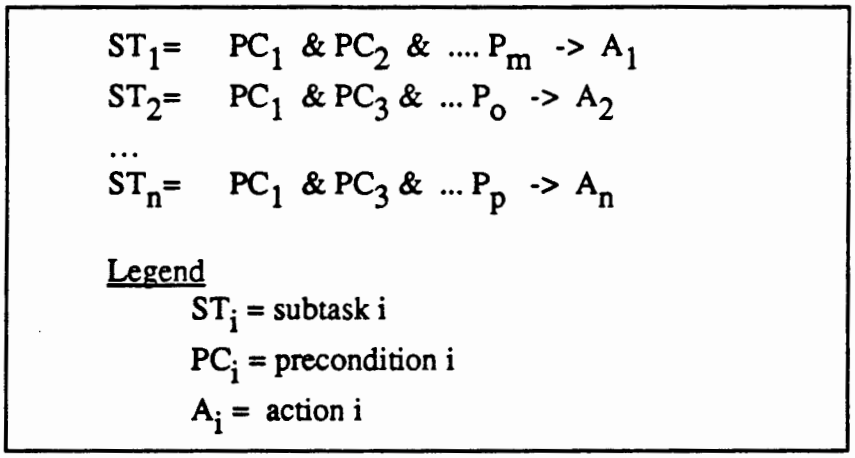

Figure 3. Subtask Definition

With this model, goal-directed behavior can be programmed as a set of "if then" triggered rules as in a production system [17]. Given the appropriate state and input command , the corresponding action is determined by matching a series of preconditions in a rule that has been programmed into the system. In effect, a state transition machine is created based on goals, preconditions, and actions. The machine moves from state to state depending on its goal, the outside world and how it was programmed.

2.2 Response Time

Control systems cover a broad spectrum of applications including flight control systems, chemical plant control, nuclear power among others. Typically, these control systems are distinguished by the importance of the reliability of the control systems operation. Any failure by these control systems can be disastrous. However, mere functional correctness is not sufficient. System performance is evaluated by the time delay required to respond (i.e. calculate) a solution. A real-time system must satisfy real physical time constraints.

Correspondingly, the basic system event is the real-time clock.

Performing task decomposition is a function of both the state of the world, and the number of operations that must be performed. This can get quite complex. Assuming infinite resources, every possible condition could be considered when making a decision. Finite computing resources limit the problem to a realistic domain. Determining the amount of work a task must perform is a subjective issue but depends heavily on the amount of time required to make a decision. Responding to an event too late nullifies the control no matter how intelligent the subsequent action. This timing restriction leads to the definition of *response time* as the maximum allowable time duration between an event and an action resulting from the occurrence of that event.

The concept of response time or hard real-time must be contrasted to soft time that is used as a sequencer of events. For example, a sequence of robot commands *GOTO A, GOTO B*, does not contain any timing information. Within the decomposition of this command, only the lower levels are worried about timing so that constant updates to the robot are guaranteed. However, the command *GOTO A BY* t_i *GOTO B BY* t_{i+1}. where t_i and t_{i+1} are some explicit time values, demands real time handling. This timing imposes on the level that the command sequence have a solution within an exact time quantum. This embodies the distinction between time as a dynamic real-world parameter versus time as a sequencing tool. While defining the timing requirements of levels within the hierarchy, this distinction must be considered at all times.

2.3 Timing Analysis

Systems that meet response time obligations are known as real-time systems. For a hierarchical control system to be real time, it must meet the demands of 1) the response time of the system plus 2) the response time of each level in the hierarchy. In the following sections, it will be assumed that each level in the hierarchy is initially assigned one processor. Then, depending on the characteristics of the level, more processors may be assigned to a level.

2.3.1 System Response Time

System response time can be considered in two ways. One perspective it to measure

the elapsed time a system takes to respond immediately. Another way to view system response time can be measured as the time required to provide a complete solution. In a chess program, this distinction can be shown when considering the difference between an immediate "thinking" response versus the time elapsed until an actual chess move. In this paper, system response will be defined as the amount of time until an immediate response. This definition leads to calculating the system response time as the sum of the worst case time responses as information (commands/status) filters up or down the hierarchy and can be characterized as follows. If n denotes the number of levels employed in the hierarchical system and the maximum response time of a control cycle of any of the levels is R_{max}, then if the ith level spends t^i_c seconds communicating/waiting for commands and status, and t^i_{proc} processing, this leads to the following minimum timing constraint for each cycle:

$$((t^i_{proc} + t^i_c) = R_i) < R_{max} : \text{for each } i=1 \ldots n$$

Because of the need for a system real-time response, an upper bound to the system response time R_s can be calculated as follows:

$$R_s < n R_{max}$$

To determine a specific system response time this implies that R_s/n defines R_{max} the maximum response time of any interval. No level can exceed this time limit and still guarantee proper system communication flow. This implies a communications heartbeat in each level must periodically sample the command and status so that updated information can filter up/down the levels. This periodic real-time sampling prevents any level from processing in isolation and skewing the system response time.

At what rate the communication heartbeat is sampling new commands and statuses, is dependent on the level within the hierarchy. If the lowest level update rate maintains R_i equal to 1 millisecond, then requiring this as R_{max} the response time limit for all other levels to sample may require too much context-switching overhead. Instead, only levels that

generate an update at the physical machine update rate are tied to this timing constraint. All higher levels that are not tied to the machine update rate may want to sample within some other reasonable time period. This leads to a definition of R_s, system response time, as the sum of R_i response times tied to the physical machine update rate, and the sum of R_j response times arbitrarily set to meet the system response time goal.

$$R_s = \Sigma R_i + \Sigma R_j \quad \text{for} \quad i=1 \ldots m \text{ and } j= m+1,n$$

$$\text{where} \quad \forall (R_k) < R_{max} \quad \text{for} \quad k=1, \ldots ,n$$

From a practical standpoint, the response time of a system dictates the amount of processing power a system needs. A system that requires a response in 10 minutes has much different requirements than a system limited by 1 secondresponse times. For example, ten minute response times provide sufficient leeway for choosing among alternative trajectory paths, but may be insufficient for planning a complicated task requiring coordination of several robots, tools and machines. Further, system response time may have no impact at lower levels. Even with a 10 minute system response time, at the lowest levels the response times may still require millisecond updates. System response times that are on the order of a second may be doable, but may require too much money and effort to be worthwhile. In effect, system response time should be as small as possible without being unrealistic.

2.1.1 Low Level Timing Requirements

The system response time leads to corresponding maximum level response time that all levels must meet. Meeting this timing restriction may not offer some levels enough processing time for sophisticated control. This leads to the motivation to divide task decomposition into planners and executors. Now, each level can maintain real-time control, yet concurrently evaluate alternative future actions. The *planner* is responsible for generating a plan consisting of a series of actions. The "best" plan is selected from a candidate list of alternative plans that achieve the commanded goal, given the current state of the environment. An *executor* enables state transitions and so is responsible for stepping

through a generated plan. The executor matches the current state of the machine against a set of preconditions as in a production system, which triggers the corresponding action. An initial assumption is that t^i_{exec}, the time the executor runs each cycle is less than the level response time limit maximum; otherwise the system response time must be increased or a finer resolution of task decomposition must be attempted.

$$t^i_{exec} < R_{max}$$

Response time for any level i, previously defined as R_i, depends on the timing constraints of the level in the hierarchy. R_i is composed of t_{exec}, the time each level spends running the executor, t_{comm}, the time spent communicating/waiting for data, and $t_{residual}$, the time remaining before the next cycle. This leads to the following definition for R_i response time for the ith level.

$$R_i = t^i_{exec} + t^i_{comm} + t^i_{residual}$$

Some general observations are in order. Obviously, $t^i_{exec} + t^i_{comm}$ must always be less than R_i since the least $t^i_{residual}$ can be is zero. This implies that if the response time of the level is very small, say 1 millisecond, this leaves very little time to do both processing and communicating. Finally, $t^i_{residual}$ the amount of residual time per level dictates whether the executor, the planner and other processes can run concurrently as interleaved processes on one processor or must run in parallel on different processors. Each of these observations will be explored further.

At the lower levels, operation is relatively independent of whatever the higher level task is executing. Control is characterized by the execution of the same or nearly the same task every cycle. Performance must be fast and predictable. An example is the servo loop at the motor level. These operations must operate with short response times and guaranteed

update rates. For example, in order for a robot to exhibit smooth motion, motion control updates to the arm must be supplied within a set time linked to the physical hardware capabilities. Control that does not meet this timing requirement results in a robot displaying jerky, stop-start motion. Thus, if a control cycle is sufficiently fast, the system will provide motion control that will appear continuous and thus in real-time. This is realized as an efficient input-compute-output cycle that maintains a small standard deviation each cycle with an upper bound σ on variance. This implies that not only must residual time $t^i_{residual}$ be greater that zero, but must be bounded by a sum of the worst case times for both processing and communication. This leads to the following constraint.

$$0 < t^i_{residual} + \sigma < R_i - t^{i,max}_{exec} - t^{i,max}_{comm}$$

$$\text{where } t^{i,max}_{exec} = \max(t^{i,j}_{exec}) \qquad j = 1,\ldots \infty \text{ cycles}$$

$$t^{i,max}_{comm} = \max(t^{i,}_{comm}) \qquad j = 1,\ldots \infty \text{ c;ycles}$$

This relation sets a limit on the $t^i_{residual}$ time available to other background processes each cycle. This further implies that adding further capabilities with a small $t^i_{residual}$ residual time will require additional processors, which will adversely affect t^i_{comm}, the time for communications.

In practice, response times at the lowest levels are very small. This leaves little time for communication and planning. Using a percentage of 10% communication time, a 1 msec response time level should allow for no more than 100 usec for all communications. A typical memory data move instruction using microprocessors is of the order of 1 usec. This allows a total of 100 data moves, with no time left for disk access, context switches, or other functions typical of a sophisticated operating system. This implies that the physical implementation of the channels of communication is of utmost importance. High-bandwidth channels are required, such as a common bus structure or dedicated links between processing nodes, with very little interface to the user. Thus, low level control will not

exhibit elaborate and/or sophisticated processing unless a parallel software algorithm is mated to special-purpose fine-grained hardware to allow more processing power per cycle. There are numerous examples that illustrate the fine-grained solutions at this level in the hierarchy [7, 11, 16, 18, 20].

2.1.1 Higher Level Timing Requirements

Moving higher in the hierarchy, alternative courses of action and real-world interaction combine to increase the complexity, although R_i, level response time, also increases. Higher levels are allowed more time to "think" about solutions. The increase in R_i means that the communications and executor heartbeat is a smaller percentage of the levels processing activity. More time per level allows a higher level to exercise more options such as secondary data storage, multi-tasking, and character input/output for a user interface. In fact, loosely coupled machines could conceivably be tied together in a distributed control system at the very highest levels for systems with a moderating response time.

In the middle of the hierarchy the concern for processor power remains. Allowing a planner and executor to run on the same processor is desirable for its simplicity and the ease on communication requirements. Whether a control level can support the executor and planning processes running interleaved on one processor or must use different processors in parallel depends on the constraints of timing and available computing resources.

In order to analyze the relationship between the planner and executor in an interleaved processing environment, several assumptions need to be established. First, levels execute every cycle and these cycles are of fixed time length c. This leads to the following equality of time cycles throughout the system.

$$t_{i+1} = t_i + c \qquad i = 0,\ldots \infty \text{ cycles}$$

Second, once the executor has completed one cycle, it must wait for the next clock cycle before processing again. It is preferred for the executor to block and wait until the next time cycle before processing at higher levels. (For lower levels, the executor could busy/wait and poll on a system clock awaiting the next clock cycle in cases where the time

for a context switch between tasks may be too large a percentage of cycle operation.)

Third, the concept of blocked, running, waiting processes implies the use of multitasking. In order to achieve real-time performance, the multi-tasking scheduling need not be fair, as in a general-purpose operating system. Instead, the multitasking must allow assignment of priorities to tasks. The requirement for priorities leads to the assumption that the executor is of higher priority than the planner. The higher priority of the executor insures that the system response requirement is preserved.

Fourth, communication between the planner and the executor uses some means that involves non-interruptible critical sections. This is important since some multi-tasking systems lack a non-preemptive feature.

Given these assumptions for interleaved execution, further timing constraints are imposed including context switching overhead, critical section support, and general planner throughput requirements. It is still to be determined whether interleaved execution is advisable. First, the planning phase of each cycle that is provided only a small $t^i_{residual}$, residual time processing would preclude a planner from formulating a plan in a reasonable amount of time. The requirement that the planner must finish any plan within d cycles should be imposed on a level. This leads to the following general constraint that a plan would take d^i cycles to finish for the ith level, each cycle using $t_{residual}$ minus two times the t_{cs} a context switching amount of processing time.

$$t^i_{plan} / d^i \quad < \quad t^i_{residual} - 2\, t_{cs}$$

If this condition cannot be held, then a multiprocessor planner/executor level should be used. Assuming the condition to finish a plan within d^i cycles does hold, more detailed constraints need to be resolved. Response time needs to be reevaluated. Therefore, including the planner on the same processor adds two extra context switches to a cycle, represented in time t_{cs}, plus increases the amount of communication time to now include

both interlevel communication $t^i_{il\text{-}comm}$ and communication between the planner and executor $t^i_{e.p\text{-}comm}$. This has the corresponding effect on response time.

$$R_i = t^i_{exec} + t^i_{comm} + t^i_{plan} + t^i_{residual} + 2\, t_{cs}$$

where $t^i_{comm} = t^i_{il.comm} + t^i_{e.p\text{-}comm}$

Since the planner and executor modules share data, whenever the planner must update the plan graph, it enters a critical section whereby it cannot be preempted. This update may or may not occur every cycle but it must be accounted for in the worst case scenario. A problem may arise if the planner is required to update a large amount of data because the amount of time allotted for the critical section must only be a fractional amount of the time used by the executor each cycle. This leads to the constraint that the planner to executor communication is small.

$$t^i_{e.p\text{-}comm} << R_i$$

If the planners cannot meet this constraint, a parallel planner and executor should be used and interprocessor communication between the processes should use a scheme that incorporates time slices or double buffers the update of the plan. If this constraint is met, one cautionary note is advised. Should a critical section in one cycle overlap into the next cycle , the executor would wait until the planner was done updating a portion of the plan graph before continuing processing. Thus, the availability of non-preemptable multitasking is required so that the scheduler does not preempt the lower priority planner in its critical section. Summarizing, the following features must be available for an interleaved planner/executor.

- high priority processes (i.e. executors) run every cycle at a fixed interval responding to current control requirements, and

- lower priority processes (i.e. planners) run in the background anticipating future control requirements.

- low priority processes allow critical sections to run to completion (i.e. non-preemptive) but must be only a fractional time portion of higher-priority processes operating cycle

2.4 System Architecture

The model for hierarchical control that has been developed exhibits concurrent system operation that can be implemented as a blend of multiple processors and interleaved execution on a single processor. Because of the disparity of response times required at various levels in the hierarchy, different levels of granularity are required. At the lowest levels, planning may be impossible, and even execution may require multiple processors to achieve a solution. Moving higher in the hierarchy, timing constraints prevent the planner and executor from residing on the same processor. In this case, the two processors would run in parallel and asynchronously the planner would update plans. At higher levels, completion time for plans is less critical; so that multitasking on a single processor can supply enough processing power for both the planner and executor. Further, if the timing constraints are not stringent, multiple levels can be combined onto one processor, or the use of a local area network[1] can be used to connect other computers as a part of the controller. Each of these configurations is based on the response time requirement of the level. Figure 4 provides guideline for the type of performance required of levels in the hierarchy.

[1] assuming a deterministic LAN that can guarantee system response

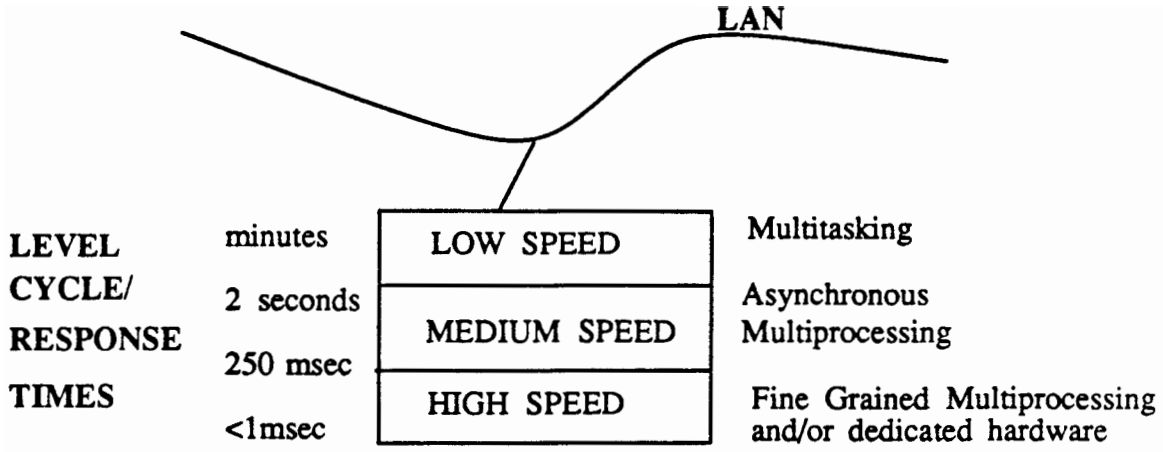

Figure 4. Computing Resource Utilization Based on Cycle Response Time

3.0 Control Level Template

A model for task decomposition within a hierarchical control system has been established that assumes multiple processors operating in parallel efficiently communicating commands and status up and down the hierarchy. Each level must have at least one executor and planner running concurrently in order to account for both the future and the present. These two concurrent processes communicate in some fashion, either via shared memory or by passing messages. The executor will read a plan each cycle. This plan may or may not be updated each cycle. The executor runs at fixed time intervals so that some clock synchronization primitives must be available. The assumption is made that the executor does not exceed its fixed time interval in a cycle and that it waits until the next time interval before processing again. The planner on the other hand is running continuously.

These capabilities form the basic template for a control level. What communication and synchronizing primitives the system use is not as important as the high-level coordination scheme. The following high level algorithm gives an overview of communication between levels within the control hierarchy. The algorithm is written in pseudo code using Brinch Hansen primitives cobegin and coend for concurrent execution [9]. It is important to note that the concurrent sections of code could be parallel on separate processors or interleaved on a single processor as dictated by the timing constraints. In this algorithm, the familiar file operations open, read, close will be used to implementcritical sections whereby no other process is modifying the data. The open corresponds to a lock, and a close corresponds to an

unlock primitives. Comments in the following algorithm will be delimited by double quotes.

```
procedure level()

cobegin
        repeat     "executor section, runs at a higher priority"
                wait_until( next-cycle);          " timing synchronization primitive "
                update(next-cycle);               "update next cycle count"
                open, read_command, close;        "  read command "
                open, read_status, close ;        "  read status "
                read plan;                        "read current plan"
                process;                          " execute level "
                open, write_command, close;       "write command "
                open, write_status, close;        "  write status "
        until forever;

        repeat  "planner"
                plan;                             "do level planning"
                open; write_plan;  close;         "update executor plan"
        until forever;

coend
```

This template for a control level is implementation independent. However, two fundamental capabilities must exist for a well-designed system.

1) The ability to communicate or share data between the levels is mandatory. The open, read/write, close operations must be from either agreed upon shared memory location or an agreed upon mailbox for message passing. Because the higher level update rates may not be tied to the machine update rate, the lower levels may be running considerably faster than the higher level executor cycles. A time stamp is required to correlate and synchronize messages. Further, in a message passing scheme this would require either that a new message arrive each cycle, or the message passing routines have the capability to check for a new message, and if none, assume to use the old message for the next cycle (or when feasible to interpolate until the next message).

2) Some form of a common memory manager that maps logical names into physical

addresses for interprocessor communication must be available. With this capability, locations of command and status buffers or message exchanges are symbolic. Levels can be added one by one to the system without use of a linkage editor to resolve inter-processor naming of either command and status buffers. This level of abstraction between the machine and the software improves flexibility and allows code to be processor independent.

Whether to perform this logical mapping statically (at compile time) or dynamically (at run time) depends on the system performance requirements. Whenever the mapping is performed the software would be best served by a consistent approach across levels. Static translations from logical to physical address typically done at linking and loading would be best for systems with very tight timing requirements. However, this precludes the opportunity for any load balancing or movement of processes within the system, since the system would have hardwired the logical connections. Dynamic mating of a logical to physical address is more robust but requires extra overhead that may not be available.

3.2 Algorithm Timing Analysis

The timing constraints of the control level algorithm outlined in the previous section can be studied using a Gantt chart notation. For the sake of accounting, the algorithm will be divided into functions. These functions will be assigned arbitrary timing constraints for the purposes of analysis. *R* will represent the function for an interprocessor read of a command and status and will take 1 ms (millisecond). *W* will represent the function for an inteprocessor write of a command and status, and will take 1 ms. *X* will represent the current level executor processing and will range from 1 to 3 ms computation intervals. *P* will represent the planner function and will require 1 to 6 ms before updating a plan. Plans must be updated within 16 ms or at worst an updated plan should be available every third cycle. *U* will represent the critical section where the planner updates the current plan graph. For simplicity, operating system computation costs such as context switches will be ignored. Initially, a response time of 8 ms will be considered. For this case, all the functions can run interleaved on one processor in one cycle, with sufficient residual processing time as shown by the Gantt chart in figure 5.

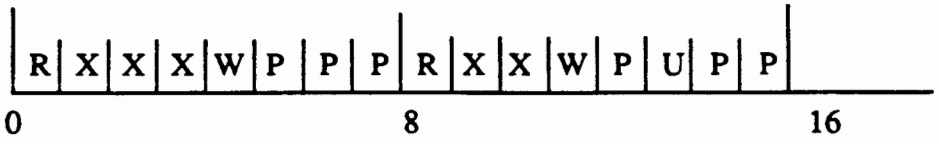

Figure 5. Gantt Chart Illustrating a Control Level Computational Pattern from Executing Interleaved on One Processor

Notice that the algorithm repeats ad infinitum the basic "RXWPU" computational pattern. The planner runs at least 3 ms each cycle which implies that after two cycles 6 ms will have been alloted to the planner with 1 ms for plan updating. More stringent timing constraints may degrade system performance beyond an acceptable level. For example, a 6 ms response time allows insufficient time for planning, so that within 3 cycles no updated plan exists. Figure 6 shows the Gantt chart for this situation.

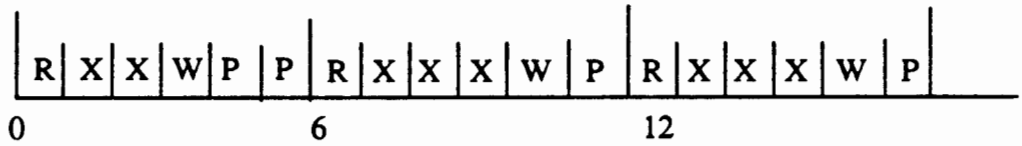

Figure 6. Gantt Chart Illustrating a Control Level Failing to Meet Timing Constraints

The lack of computational resources for planning suggests the partitioning of the problem across processors. Now, processor one (*P1*) will have exclusive execution, but will have the additional timing constraint of interprocessor communication with the planner. This interprocessor communication overhead is represented by the function *C* and requires 1 ms. With excess computational power, the procesor must now idle between cycles represented by a dash (i.e. -). The executor repeats ad infinitum a "CRXW-" computational pattern. Processor two (*P2*) will execute planning exclusively. Computational resources easily guarantee 6 ms planning to generate updated plans with 12 ms. As a side effect of excess computational power, the planner may have to wait for fresh information from *R*, an executor status read. (The shared *R* status information is assumed to be in a critical section and read-

only for the planner.) The Gantt chart in figure 7 shows the execution scheme in a parallel processors.

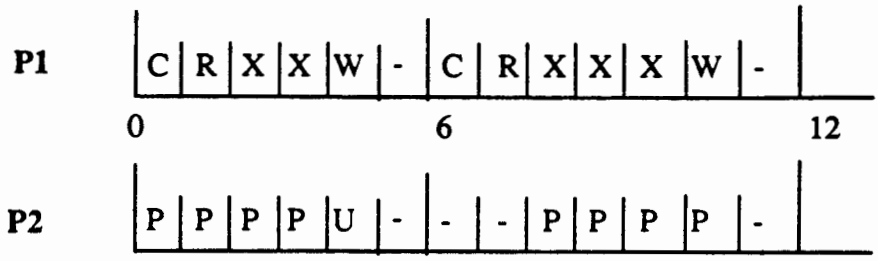

Figure 7. Gantt Chart Illustrating a Control Level Computational Pattern from Parallel Execution on Two Processors

In the parallel design, the control level effectively meets all timing constraints with some computational waste. However, this solution can be sensitive to increases in complexity of the executor (*X*) portion of the control cycle. For example, if the function X is replaced by a newer version which requires 4 ms a cycle, then the above functional allocation across processors will not work. For in the worst case, there is no time for interprocessor communication with the planner (*C*). In this case, faster hardware must be bought or a more refined task decomposition strategy must be used.

4. Conclusion

This paper has addressed some design and timing issues associated with developing a hierarchical control system for an intelligent machine. Hierarchical structuring for a control system creates levels of parallel operation that can be characterized as virtual control loops. Virtual control loops are a software model of a machine duty cycle where software emulation samples input command from a neighboring upper level, compares this to the sensory sampled environment, computes a goal directed output, and output an action to the neighboring lower level, all within a fixed response time.

Mapping this hierarchical control system onto a parallel pipelined computing system is a efficient and practical method of implementation. It offers the flexibility for inter-level communication at a reasonable cost. Levels in the hierarchy are primed with commands and then local control flow within each level proceeds independently of neighboring levels. The

global control flow is still goal directed because each level is periodically sampling new commands and reporting status to neighboring levels. A system response time is guaranteed because of this periodic sampling of commands and status.

Task decomposition is used to create the level of granularity for interlevel module definition necessary to meet the strict real-world timing requirement at the lowest level while offering a convenient software structuring tool at the higher levels of control. Information hiding between higher levels results and allows well-defined, standardized interfaces to be developed. Task decomposition control allows systematic software structuring that offers predictable timing behavior that can be modeled as the sum of the processing plus inter-processor communication costs. Within each control level, a planner and executor run concurrently; the planner accounting for future action, and the executor handling real-time responses. The planner and executor can run concurrently on the same processor given sufficient residual time per control cycle for planning in a pre-determined number of cycles. Adding more processors to a level will allow a level to meet the timing constraints but will adversely affect the communication time per level, as well as increase the complexity.

This hierarchical pipelined parallel control system exhibiting virtual control loops has been implemented with a purely executor style of task decomposition for a robot control system at the National Bureau of Standards [3]. The flow of control was based on state-table transitions. Where planning was appropriate, static plan definitions were used. The system offered several benefits. First, the system was sensory-interactive and adapted to perturbations in the environment in real-time even though the world model was limited to basic feature recognition. Second, hierarchical decomposition created well-defined interfaces that allowed substitution of different implementations of a level with little effect on the higher or lower levels that lead to proposed interface standards [6]. Finally, the system was able to execute in real-time and supply millisecond robot updates while acounting for perturbations in the environment.

Work has begun on implementing a full hierarchical parallel pipelined control system that includes planning, extensive world modeling including maps, object definitions and

feature recognition [2]. The hierarchical control systems under development will be adapted for both robot and autonomous vehicles. To maintain cost, flexibility and portability, multiprocessors communicating across a shared backplane has been chosen for the parallel architecture [21]. Eventually the cost of fine-grain machines may allow a process per processor for the design of a robot control system; in the meantime a hybrid system that supports fine and medium grain processes offers a cost-effective and efficient design methodology.

References

[1] J.S. Albus, A.J. Barbera and R.N. Nagel, "Theory and Practice of Hierarchical Control," *Twenty-third IEEE Computer Society International Conference*, pp. 18-39 1981.

[2] J.S. Albus, H.G. McCain, and R. Lumia , *NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, Md., July 1987.

[3] A.J. Barbera, M.L. Fitzgerald, and J.S. Albus, "Concepts for a Real-Time Sensory-Interactive Control System Architecture," *Proceedings of the 14th Southeastern Symposium on System Theory*, April 1982.

[4] V. Dupourque, H. Guiot, and O. Ishacian, "Towards Multi-processor and Multi-robot Controllers," In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal.), IEEE, New York, pp. 864-868, April 1986.

[5] M. Dubois, C. Scheurich, and F.A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, pp. 9-21, Februrary 1988.

[6] M.L Fitzgerald, A.J. Barbera, and J.S. Albus, "Real-Time Control Systems for Robots," *SPI National Plastics Exposition Conference*, 1985.

[7] D. Gauthier, P. Freedman, G. Carayannis, and A.S. Malowany, "Interprocess

Communication for Distributed Robotics. *IEEE Journal of Robotics and Automation*, Vol. RA03, No. 6, pp. 493-504, Dec. 1987.

[8] R.D. Gaglianello, and H.P. Katseff, "A Distributed Computing Environment for Robotics," In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal.), IEEE, New York, pp. 1890-1895, April 1986.

[9] P.B. Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[10] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, August 1978.

[11] J.U. Korein, G.E. Maier, R.H. Taylor, and L.F. Durfee, "A Configurable System for Automation Programming and Control," In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal.), IEEE, New York, pp. 1871-1877, April 1986.

[12] C.P. Kruskal and C.H. Smith, "On the Notion of Granularity", National Bureau of Standards Report, July 1987.

[13] G.E. Lyon, "On Parallel Processing Benchmarks", National Bureau of Standards Report , NBSIR 87-3580, pp. 1-23, June 1987.

[14] G.E. Lyon, "Programming The Parallel Processor", In *Second Symposium on the Role of Language in Problem Solving*, sponsored by the Applied Physics Laboratory of Johns Hopkins University, April 2-4, 1986.

[15] R.B. McGhee, D.E. Orin, D.R. Pugh, and M.R. Patterson, "A hierarchically-structured system for computer control of a hexapod walking machine," In *Proceedings of the 5th IFTOMM Symposium on Robots and Manipulator Systems*, (Udine, Italyl, June 1984). IFTOMM, 1984.

[16] S. Narashimhan, D. Siegel, J.M. Hollerbach, K. Biggers, and G. Gerpheide, "Implementation of control methodologies on the computational architecture of the Utah/MIT hand," In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal.), IEEE, New York, pp. 1884-1889, April 1986.

[17] N. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, CA, 1980.

[18] R.P. Paul, and H. Zhang, "Design of a Robot Force/Motion Server," In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Cal.), IEEE, New York, pp. 1878-1883, April 1986.

[19] D.L. Parnas, "On a 'Buzzword': Hierarchical Structure," In Proceedings of the IFIP Congress, North-Holland, Amsterdam, 1974.

[20] K. Schwan, T. Biharit, B. Weide, and G. Taulbee, "High-Performance Operating System Primitives for Robotics and Real-Time Control Systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 3, pp. 189-231, August 1987.

[21] A. Topper, and V. Hayward, "Porting RCCL to a Multiprocessor Environment : Requirements Specification For Hardware and Real-time Software," McGill Research Center for Intelligent Machines, 1987.

List of Figures: