Timed Perturbation Analysis: An Approach for Non-Intrusive Monitoring of Real-Time Computations *

Madalene Spezialetti
mspezial@csee.lehigh.edu
EECS Dept., Packard Lab
Lehigh University
Bethlehem, PA 18015

Rajiv Gupta
gupta@cs.pitt.edu
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

A task which is a part of a real time application must not only perform a specific function but it must also execute under timing constraints specified in the form of deadlines by which the output results must be computed. Thus, any attempt to monitor the run-time behavior of a real time task through code instrumentation can potentially alter the program's timing behavior. In this paper we develop perturbation analysis techniques to identify the situations in which the run-time monitoring activities can be performed non-intrusively. The techniques identify the idle time available during the execution of a task and schedule the monitoring task during these times. Instead of treating a monitoring task as an indivisible unit, we partition the monitoring work among various points at which idle time is available.

Keywords - monitoring, tracing, instrumentation, perturbation analysis, asynchronous communication.

1 Introduction

Due to the complexity of both sequential and distributed real time applications, techniques must be created which aid in the development and maintenance of these programs. A variety of techniques have been developed for use on non-real time applications. One such technique is the monitoring of program behavior during execution. In this approach, the application program is instrumented, at various points, to save specified information. This information can then be sent to a remote site at which it can be stored to trace program behavior, analyzed or displayed. Monitoring requests will take two forms: a user may request to capture a certain portion of the local state at a specific program point; or a user may request the continuous tracing of a certain state, such as a variable, throughout the execution. A request for tracing can be viewed as multiple monitoring requests of the former kind.

The task of monitoring can be relegated to a single dedicated process or a group of processes referred to as monitors. The processes belonging to the computation are then instrumented to communicate the relevant information to the monitor. Alternatively, a portion of the monitoring code could also be incorporated directly into the application modules. However, any approach based upon instrumentation is intrusive, that is, it can alter the behavior of a program in some respects. In a real time application the problem of intrusion is even more pronounced than it is in non-real time programs, since delays resulting from the execution of monitoring instrumentation may cause deadlines in real time programs to be missed. Without the aid of specific hardware enhancements, delays due to the execution of monitoring code will invariably occur. It is the goal of this paper, however, to provide techniques to analyze real time programs to determine if and when monitoring-related delays can be absorbed by the application program such that the delays will not cause any deadlines to be missed.

*Supported in part by the National Science Foundation through Grant CCR-9212020 to Lehigh University and a Presidential Young Investigator Award CCR-9157371 to the Univ. of Pittsburgh.

In order to carry out such analysis, the real time application can be viewed as a series of execution spans, delineated by input points, at which the computation must wait to receive data. *Idle times* occur during those periods when the current execution is complete and the computation is suspended at an input point. Thus, the idle time can be viewed as the amount of monitoring work that can be absorbed without effecting the program's ability to meet deadlines.

To aid in the introduction of monitoring instrumentation into an application, software support must be provided. The user would indicate a particular monitoring activity that should be carried out. Automatic techniques will be used to perform the analysis for determining if the monitoring overhead could be absorbed. A straightforward technique would be to determine if the idle time of the span in which the code is to be inserted is greater than or equal to the execution time of the monitoring code. Based on this comparison, the user would be informed whether or not the code could be non-intrusively added.

This approach, however, may deny the inclusion of monitoring instrumentation unnecessarily, due to the fact that the instrumentation is viewed as an indivisible unit and only the idle time within the span where the monitoring point is specified is considered. However, both of these restrictions are not necessary. First, the monitoring task itself can be decomposed into subtasks, which may include the saving of data and further transmission of the monitoring of data. Often, these tasks may be divided and need not be carried out as a single unit. Further, idle time itself may exist at numerous locations throughout the application. Therefore, it may be possible to divide the monitoring task into subtasks and distribute these subtasks to take advantage of idle time at various points in the application. In this way, it may be possible to non-intrusively absorb more monitoring instrumentation then would be possible using the straightforward approach.

It is therefore the goal of this paper to perform timing analysis on sequential or parallel real time programs to determine one or more program points at which monitoring instrumentation can be placed so that it will be non-intrusively absorbed by the application.

Extensive research has been carried out in the area of real-time languages and their analysis for scheduling [4, 14, 18]. Previous work on timing analysis has mainly concentrated on execution timing analysis. This includes compiler support for computing WET estimates [9, 10, 12] and the run-time refinement of WET estimates based upon a combination of compile-time information [15, 6] and run-time information [3, 5]. In [19] authors present the straightforward approach of introducing the monitoring activity at the user selected points and [20] presents a hardware solution to non-intrusive monitoring by providing architectural support. Our approach can enhance the effectiveness of monitoring systems such as the ISSOS [13].

In the next section we will present the models used to characterize the real time application program and the monitoring tasks. In section 3 we present timing analysis algorithms by which monitoring tasks are integrated into a real time application task. In section 4 we present extensions to our approach to handle parallel real time applications which are composed of multiple tasks. We conclude with a brief discussion of work in progress.

2 Real-Time Tasks and Monitoring Tasks

A typical real time application can be modeled as a repeated execution, at regular intervals, of a code segment. For example, the *Infrared Missile Warning* application encountered in avionics receives an image represented by a 2D mesh of pixels as the input. This image is processed through various spatial and spectral filters and then examined for the presence of missiles. This task must be completed by a specified deadline following which a new image is received and the task is repeated all over again. The code segment that implements the real time task is characterized as containing input points, output points and deadlines. At the input points information

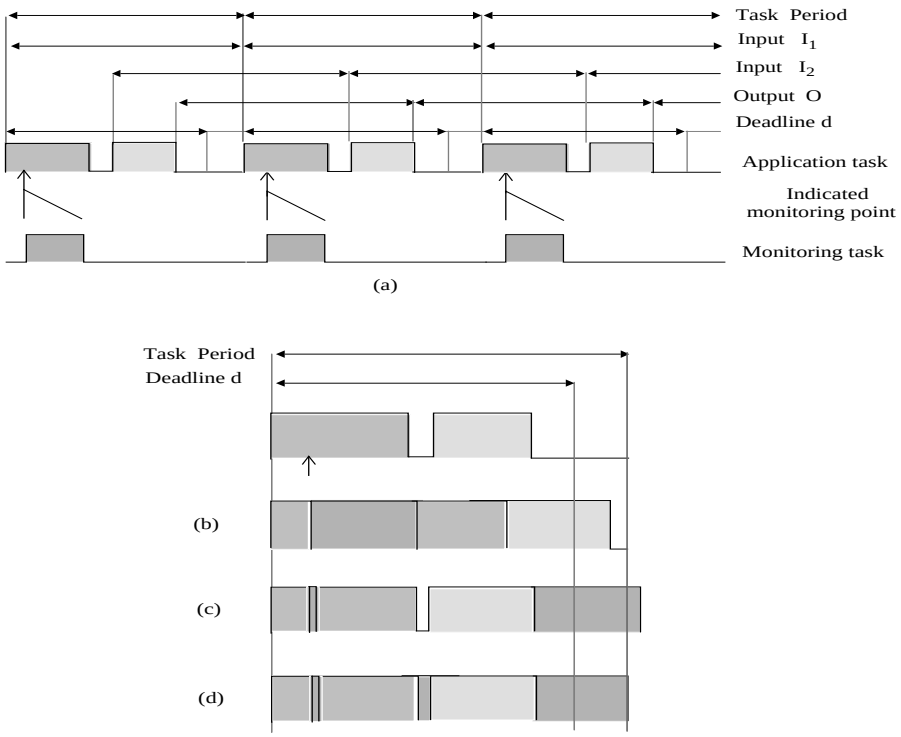


Figure 1: Introducing Monitor in a Real-Time Task

is received by the application and it is assumed that the execution can not progress past such a point until the information has been received. A period during which the execution of an application is suspended due to the need to await the arrival of input is termed *pre-input idle time*. The output points represent those points at which an expected result is generated. Deadlines are associated with output points, and the expected result of the output point must be generated by the associated deadline. There may exist idle time between successive executions of the application task if the first input needed to begin the task is not yet available. Such a period is termed *post-completion idle time*. It is the segments of pre-input and post-completion idle time which will be used to non-intrusively absorb monitoring activity.

As an example of an application task, consider the diagram in Figure 1. In this application, the main code segment, which will be executed repeatedly, will accept two inputs I_1 and I_2 at regular intervals, and produce a single output, O, with a deadline, d, associated with it. The implementation of the real time application results in the timing behavior shown in Figure 1. A segment of code is executed after the reception of the first input I_1. However, this code segment completes execution before the arrival of the second input, I_2. This situation leads to pre-input idle time associated with I_2, during which the execution of the application must suspend until the new input is received. Following the generation of the output, O, there is a post-completion idle time before the next iteration of the task can begin.

Let us now consider a number of approaches to accommodating monitoring activity into the application task based on the specification of a monitoring point by the user. The straightforward approach would be simply to insert the monitoring task directly at the user specified point, as shown in Figure 1b. However, this technique has a number of drawbacks. By placing the code directly at the specified point, the monitoring task is essentially

given a higher priority than the application task, since any processing of the application task will be postponed until the monitoring task is complete. As a result, deadlines may be missed as shown in Figure 1b. Conversely, if the determination of whether or not the monitoring task should be inserted is based solely on whether or not the next deadline can be met, monitoring insertion may be denied even though it could potentially have been accommodated by breaking and distributing the monitoring activity.

Next let us consider approaches in which we split the monitoring task. The local saving of data that takes little time, as it entails only the storing of data at a specified location, is performed at the monitoring point. The packaging and transmission of the monitoring data which dominates the local monitoring actions can be inserted at a later point in a task. Thus, a second approach to monitoring insertion would divide the monitoring task to insure that the monitoring task does not take priority over the application task and cause deadlines to be missed. This approach would place only the segment of code needed to store monitored data at the insertion point, and then only carry out the more time-consuming portion of the monitoring task during post-completion idle time. In this way, the monitoring task is given lower priority in relation to the completion of the application's computation. However, it may not be possible to carry out the monitoring activity before the expected arrival of the input data needed to begin the next iteration of the application. This scenario is shown in Figure 1c. In this case, the monitoring activity could be abandoned and not completed or, if it were completed, a delay would occur in beginning the next iteration.

Finally, let us consider a third approach, in which the monitoring task is broken up and can be carried out at a number of points. At the insertion point specified by the user, the required data would be saved locally. The more time consuming transmission, however, will be distributed, if necessary, to be carried out in portions at various times. When determining where the remaining portion of the monitoring activity should be carried out, the post-completion idle time would be utilized first, thus insuring that the application itself will have the highest priority. If the task can not be completely accommodated in post-completion idle time, pre-input idle times will be utilized to carry out portions of the monitoring task. This approach will insure that monitoring activity will only be inserted at points where it can be handled non-intrusively, that the monitoring does not take priority over the application and that, by attempting to insert portions of the monitoring task during post-completion and pre-input idle times, more monitoring activity can be accommodated non-intrusively than would be possible if the monitoring task were treated as an indivisible unit. An example of this approach in shown in Figure 1d. It is this approach to the insertion of monitoring code which will be presented in the remainder of this paper.

3 Timed Perturbation Analysis of a Single Real Time Task

In this section we consider a real time application that is implemented as a single sequential task which is executed repeatedly, receives inputs at regular intervals, and is required to generate outputs at regular intervals to meet specified deadlines. We assume that the input and outputs points are identified in the code that implements the task and that these points are guaranteed to be visited during each iteration through the task. The following notation is used during this analysis:

- The time at which an input I becomes available is denoted as $t_{avail(I)}$. The *deadline* for an output O is denoted as $t_{avail(O)}$. The times $t_{avail(I)}$ and $t_{avail(O)}$ are measured as the time elapsed since the beginning of the current invocation of the task.

- For a stream of inputs/outputs that are consumed/produced inside a loop, the time at which the first input/output is available/produced is specified, $t_{avail(I)}/t_{avail(O)}$, and in addition the interval, $\tau_{period(I)}/\tau_{period(O)}$,

at which the subsequent inputs/outputs are available/produced is also specified.

The goal of timing analysis is to identify for each statement i the starting time, $start_i$, and the completion time, $finish_i$, of the statement assuming that the maximum execution time, $exec_i$, of each statement is given and enough resources are available to execute the tasks in parallel. In addition to the execution times of statements the analysis also takes into account the constraints associated with the task inputs. Following the analysis the amounts of pre-input and post-completion idle times are known.

This analysis is performed using the control flow graph (CFG) representation of the program instrumented with code fragments for saving data requested by monitoring requests. We assume that the CFG is reducible. The analysis is based upon intervals [1] in the CFG where each interval corresponds to a program loop L. Each interval has a header node h, a last node l, and a back edge from l to h. We assume that the lower and upper bounds of the loop index variable, and thus the number of loop iterations, for each loop are known at compile-time.

The timing analysis is carried out in two passes. Two passes are required because the timing information being computed includes both *synthesized* and *inherited* attributes. The execution time of a loop is a synthesized attribute since it is computed by examining the statements inside the loop. The starting time of the loop is an inherited attribute since it depends upon the execution timing of the statements preceding the loop. Thus, an efficient approach to this analysis requires two passes. The first pass processes intervals starting at the innermost level and terminating at the outermost level. This allows the execution times of loop iterations, and hence the entire loops, to be computed. In the second pass, the starting times of various input and output statements are computed by propagating execution time information from earlier parts of the CFG to the later parts and pre-input delay nodes are introduced to ensure that no input is consumed before it becomes available.

```
1. Algorithm ComputeExecutionTimes {
2.     for each interval L, in innermost to outermost order do
3.         – initialize the header node h
4.         start_h^{low_L} = 0
5.         – process the nodes in the interval
6.         for each node in L in reverse-depth-first order do
7.             – Let Pred_i be the set of immediate predecessors of i
8.             start_i^{low_L} = MAX_{p∈Pred_i} finish_p
9.             finish_i^{low_L} = start_i^{low_L} + exec_i
10.        endfor
11.        – compute loop execution time
12.        iter_L = finish_l^{low_L}
13.        finish_h^{high_L} = (high_L - low_L + 1) × iter_L
14.        exec_L = finish_h^{high_L}
15.    endfor
16. }
```

Let us consider the first phase of analysis, presented in algorithm *ComputeExecutionTimes*, in greater detail. In the following discussion L refers to a loop and low_L and $high_L$ are the loop bounds. The times $start_i^j$ and $finish_i^j$ denote the starting time and completion time of statement i during loop iteration j, where j is the loop index variable. The starting time of the loop is initialized to zero. Next, starting at the header and terminating at the last node in the loop, the starting and completion times for all statements in the loop are computed. Following this pass over the statements in the loop the completion time of the last node in the loop essentially provides us with the execution time of a single loop iteration. From the execution time of a single loop iteration, $iter_L$, the execution time of the entire loop, $exec_L$, can now be computed. The completion time of the entire loop, $finish_h^{high_L}$, is the completion time of the header node following the last iteration. As the analysis proceeds to

outer intervals, an enclosing interval views a nested interval L as a single node with the execution time of $exec_L$, which would already have been computed.

After the completion of the above pass the execution time of each loop is known; however the times at which the various input and output statements inside a loop are executed is unknown since the code was processed from the inside out assuming that the starting time of each loop header is zero (see line 4). In the second phase the loops will be processed in the reverse direction, that is, from the outermost loop to the innermost loop. This will enable us to compute $start_h^L$ for a loop and hence the starting time of input and output statements in the loop. We will only compute the starting and completion times of statements in intervals containing input and/or output statements since this is the only information needed to identify idle times. However, our technique is general and can be used to determine the timing information for all statements.

We assume that the outermost interval representing the real-time task is a loop. If it is not a loop we can view it as a loop with a single iteration. The starting time of the outermost interval is initialized to zero. The starting time of a nested interval is computed from the finishing times of the predecessors of the header node. The starting and completion times of the nodes in an interval are computed in the reverse-depth-first order.

If a nested interval is encountered then the processing of the enclosing interval may have to be suspended until the nested interval has been processed. If the code inside the nested interval contains no input or output statements then we can compute the completion of the nested interval and continue processing. However, if the nested interval contains input statements, we must suspend the processing of the enclosing interval, since the presence of input statements may require delays to be introduced which will alter the execution time of the nested interval. We also suspend the processing of an interval if a nested interval with output statements is encountered. The presence of output statements requires processing of the nested interval so that completion times of output statements can be computed. Once the completion time of the nested interval is known the processing of the enclosing interval resumes. Finally when the entire interval has been processed we know the updated time that it takes to execute a single iteration of the current interval.

Once the starting time of an input statement is computed, it is compared with the time at which the input becomes available to determine the need for a pre-input delay. Let us consider an input I which is a single input, that is, it is executed only once during each iteration through the real-time task. If the computed value of $start_I$ is lower than $t_{avail(I)}$, then a pre-input delay of length $t_{avail(I)}$-$start_I$ is introduced immediately preceding input I. Consider a stream of inputs which are received by an input statement I that is nested inside a loop L with header h and last statement l. For simplicity let us assume that the input is received at the top of the loop. Once the time $start_h^{low_L}$ has been computed we compare this value with $t_{avail(I)}$ and if required introduce a delay, $delay_{L(I)}^{out}$, immediately preceding the loop L. Next, pre-input delay, $delay_{L(I)}^{in}$, is introduced inside the loop to meet the constraints imposed by the interval associated with a stream of inputs consumed inside a loop. The loop iteration time $iter_L$ is compared with $\tau_{period(I)}$, the interval at which inputs become available. If the comparison yields a need for a delay, this delay, $delay_{L(I)}^{in}$, is introduced along the loop back edge. Figure 2 shows where the pre-input delays are placed for an input I. The iteration execution time and loop execution time are updated if delays are introduced. The algorithm *PlaceDelays* summarizes the timing analysis. This algorithm is called with the interval representing the entire task as its parameter.

After the timing analysis described above we determined whether the current implementation of the real time task meets all the specified deadlines. If the completion time of output an O, $finish_O$, is less than the output deadline, $t_{avail(O)}$, then the deadline will be met. If deadlines are met then we compute the total idle time available for introducing monitoring tasks.

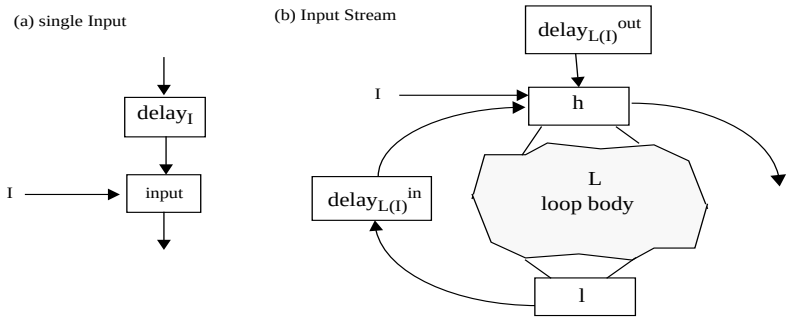


Figure 2: Introducing Pre-input Delays.

```
1. Algorithm PlaceDelays ( L: an interval ) {
2.       – initialize the starting time of interval L
3.       if L is the outermost interval then start_h^low_L = 0
4.       else start_h^low_L = MAX_{p∈Pred_L} finish_p, where Pred_L = Pred_h - {l}
5.            if L contains no input/output statements then
6.                 finish_h^high_L = start_h^low_L + exec_L
7.                 return()
8.            endif
9.       endif
10.      – introduce a delay node, if required, immediately outside L
11.      if interval L receives an input I then
12.           if start_h^low_L < t_avail(I) then delay_L(I)^out = t_avail(I) - start_h^low_L
13.           else delay_L(I)^out = 0 endif
14.           start_h^low_L = start_h^low_L + delay_L(I)^out
15.      endif
16.      – compute the starting times of input/output nodes in L
17.      for each node i in L in reverse-depth-first order do
18.           if node i is an interval L' then
19.                PlaceDelays ( L') – process nested interval L'
20.           else – node i is a statement node
21.                start_i^low_L = MAX_{p∈Pred_i} finish_p
22.                where if p ∈ L, finish_p = finish_p^low_L
23.                      if p is header of a nested interval L', finish_p = finish_p^high_L'
24.                if node i is an input node then
25.                      – input nodes in the outermost interval
26.                      if start_i^low_L < t_avail(i) then
27.                         delay_i = t_avail(i) - start_i^low_L
28.                      else delay_i = 0 endif
29.                endif
30.                start_i^low_L = start_i^low_L + delay_i
31.                finish_i^low_L = start_i^low_L + exec_i
32.           endif
33.      endfor
34.      – introduce a delay node, if required, along loop back edge inside L
35.      if finish_l^low_L < τ_interval(I) then delay_L(I)^in = τ_interval(I) - finish_l^low_L
36.      else delay_L(I)^in = 0 endif
37.      iter_L = finish_l^low_L + delay_L(I)^in
38.      exec_L = finish_h^high_L + delay_L(I)^in × (high_L - low_L + 1)
39. }
```

We determine if a monitoring request that requires the packaging and transmission of specified amount of data can be accommodated by the idle times in a schedule as follows. We assume that the time entailed in processing S amount of data is given by:

$$T(S) = a \times S + b \times \lceil \frac{S}{M} \rceil.$$

The first term models the cost of packaging data and the second term models the cost of transmitting messages of maximum size M. Thus, the maximum amount of data that can be processed during time duration t is given by:

$$T^{-1}(t) = S \ni T(S) \leq t < T(S+1).$$

We first compute the amount of data that can be processed during pre-input idle times as shown below. In this equation node i is a delay of length $delay_i$ which is encountered $count_i$ number of times.

$$PreInput = \sum_{i=1}^{n} T^{-1}(delay_i) \times count_i$$

The amount of data that can be processed during the idle time available at the end of each iteration through the task is given by the expression below, where *end* denotes the last statement in the task.

$$PostCompletion = T^{-1}(TaskInterval - finish_{end})$$

Therefore if the amount of monitoring request requires sending MON amount of data the request can be accommodated if the following condition is true.

$$\exists j,\ where\ j\ is\ after\ the\ monitoring\ point\ \ni\ total\ PreInput\ after\ j\ +\ PostCompletion \geq MON.$$

Since monitoring tasks are non-essential tasks, we will only schedule their execution if there is idle time in the schedule that can be safely used to accommodate the monitoring activity. In other words if the scheduler finds that idle time is available, it will use it to perform monitoring activities.

An example illustrating the results of timing analysis is given below. In the code for the tasks $\{compute\ for\ time\}$ and $\{pre-input/post-completion\ delay\ for\ time\}$ indicate the time periods for which computation is performed and time periods during which idle-time is encountered. The computation times represent the WET estimate of one or more statements. The *delay* nodes indicate points at which monitoring activity can be performed.

Task Period = 150;
External Input I_1: $t_{avail(I_1)} = 0$;
External Input I_2: $t_{avail(I_2)} = 15$ and $\tau_{period(I_2)} = 20$;

1. Receive external input I_1;
2. { compute for 10 }
3. { pre-input-delay for 5 }
4. **Do** i = 1 **To** 5
5. Receive external input I_2;
6. { compute for 13 };
7. **if** $i \neq 5$ { pre-input-delay for 7 };
8. **EndDo**
9. { compute for 17 }
10. { post-completion-delay of 25 }

In the discussion so far we introduced the monitoring work at points immediately preceding input statements and following task completion. Thus, the monitoring code is introduced at points which are guaranteed to be executed. If the user makes a monitoring request for tracing the value of a variable, then the amount of monitoring work may vary from one execution to the next. If the entire monitoring activity is introduced at points preceding inputs and following task completion then we must reserve enough time for maximum possible monitoring work that may be performed. A more effective approach would be introduce conditionally performed monitoring activity at conditionally executed program points. For example, if the amount of work performed during the then-part of an if-statement is less than the amount of work performed during the else-part, then some amount of monitoring activity associated with the then-part can be introduced in the then-part rather than at a pre-input point later in the task. The time saved from pre-input and post-completion idle times can be used by other monitoring tasks.

4 Analysis of Parallel Real-Time Tasks

A real time application may be too time consuming to be executed under the specified deadlines. We assume that under such conditions the application is partitioned into a communicating set of tasks that are executed in parallel on different processors on a distributed-memory system. By choosing a distributed-memory system we avoid the problems associated with predicting memory reference behavior of cache based shared-memory systems. For example, in the *Synthetic Aperture Radar* application the code for computing an eight point FFT dominates the processing time. We can speed up the FFT computation by a factor of eight by exploiting the inner loop parallelism available during the FFT computation. The analysis discussed in the preceding section can be extended to handle a parallel real time application. Each task in a parallel application can still be viewed as a computation with inputs, outputs and deadlines. Some of the inputs are *external* inputs as before and others are *internal* inputs that are essentially outputs of other tasks. We assume that the communication among tasks is achieved through non-blocking *sends* and blocking *receives*. Furthermore, it is assumed that each send operation is paired with a unique receive in another task and that the communication statements are definitely executed during each execution of a task. These assumptions are in context of a real time application. In this analysis we denote interprocess communication delay by δ_c. The interprocess communication delay is denoted by δ_c. The value of δ_c will depend on the particular architecture and also on the amount of data transmitted.

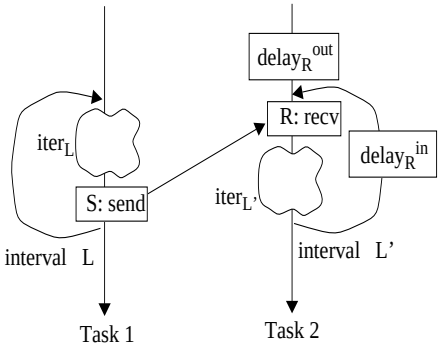
The busy-idle analysis is applied to an application containing multiple tasks as follows. We first compute the execution times of the statements in each task independently using the algorithm *ComputeExecutionTimes*. Next we simultaneously initiate algorithm *PlaceDelays* for each of the tasks. Coordination among the instances of *PlaceDelays* is necessary to process a set of communicating tasks since the timing information of the sending statement must be computed before the processing of the receiving task can continue. Let us consider the situation in which a stream of values are communicated from one task to another. If the rate at which data is being generated by the sending task is lower than the rate at which data can be consumed by the receiving task, delays will be introduced. Consider the situation illustrated in Figure 3. The receive in interval L' can be viewed as an input I whose characteristics are determined by processing interval L and these characteristics are then used for introducing delays as shown in Figure 3.

5 Implementation Issues and Architectural Support

Next we discuss the issues relating to the implementation of interleaving a real-time task and its monitoring task. so that they execute on the same processor in a coroutine-like manner. Although, the monitoring requests

1. $t_{avail(I)}$=$finish_{send}^{low_L}+\delta_c$; and $\tau_{period(I)}$=$iter_L$.
2. **if** $iter_L > iter_{L'}$ **then**
3. $delay_R^{in}$=$iter_L$-$iter_{L'}$
4. **endif**
5. **if** ($finish_{send}^{low_L}+\delta_c$)>$start_{recv}^{low_{L'}}$ **then**
6. $delay_R^{out}$ = ($finish_{send}^{low_L}+\delta_c$)-$start_{recv}^{low_{L'}}$
7. **endif**

(a)



(b)

Figure 3: Communicating Tasks.

must be provided at compile-time, it would be desirable if the actual monitoring activity can be turned on/off dynamically by activating/deactivating switching instructions. We must locate the points in the code at which the switching is to be performed, introduce the switching instructions, and then provide a mechanism for activating or deactivating appropriate instructions based upon the generated schedule. One approach that can be taken is to allow the compiler to introduce the switching instructions in the code. However, these instructions are provided with guards to make them inactive, that is, none of these instructions will be executed if the tasks are executed in their present state. In its final phase, the scheduler can use this information, to activate the appropriate switch instructions.

Architectural support can be provided to assist the interleaved execution of a task and its monitor. First, a single bit in a switch instruction can be provided to indicate whether it is active or inactive, rather than introducing explicit guard variables. Also the architecture can provide two register sets, one for each task, so that the cost of switching between the tasks is a single instruction. In addition to architectural support for switching we must also provide caching support for interleaved scheduling. By partitioning the cache among the interleaved tasks we can ensure that the cache locality is not adversely affected. One such strategy was proposed in [11].

6 Future Work

In this paper we used our timing analysis information to accomodate monitoring code. However, we can also use this information to enhance the scheduling of real-time tasks. The execution profile of a task derived from this timing analysis indicates the times at which a task is executing and the times at which it is waiting for an external input. A real-time scheduler can use these profiles to generate schedules that are able to meet deadlines by interleaving and overlapping the execution of different tasks, where no schedule that meets the deadlines may exist without interleaving or overlapping. Using the timing information we have developed a compact task graph representation which can be used to generate schedules efficiently [8]. We are currently developing extensions to algorithms for scheduling periodic tasks [2, 16] for generating interleaved schedules. We have also developed other notions of non-intrusion, and static analysis algorithms for their detection, which are applicable to non-real time applications [7] [17].

References

[1] M. Burke, "An Interval-based Approach to Exhaustive and Incremental and Interprocedural Data-flow Analysis," *ACM Transactions on Programming Languages and Systems*, Vol.12, No.3, pp.341-395, 1990.

[2] S. Cheng, J.A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proceedings of the 7th Real-Time Systems Symposium*, pp.166-174, 1986.

[3] P. Gopinath, T. Bihari, and R. Gupta, "Compiler Techniques for Generating Predictable Object-Oriented Real-Time Software," *IEEE Software*, special issue on Real-time systems, pp.45-50, September 1992.

[4] P. Gopinath, T. Bihari, and R. Gupta, "Supporting Real-Time Software Integrated Circuits," *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pp.55-61, Phoenix, Arizona, December 1992.

[5] P. Gopinath and R. Gupta, "Applying Compiler Techniques to Scheduling in Real time Systems," *Proceedings of the 11th Real-Time Systems Symposium*, pp.247-256, Orlando, Florida, December 1990.

[6] R. Gupta and P. Gopinath, "Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications," *Proc. of 11th IEEE Workshop on Real-Time Operating Systems and Software*, Seattle, Washington, May 1994.

[7] R. Gupta and M. Spezialetti, "Towards a Non-Intrusive Approach for Monitoring Dist. Computations through Perturbation Analysis," *Proc. 6th Work. on Lang. and Compilers for Par. Comp.*, LNCS 768 Springer Verlag, pp.586-601, 1993.

[8] R. Gupta and M. Spezialetti, "Busy-Idle Timing Analysis and Compact Task Graphs: Compile-time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks," *Technical Report TR-94-24, University of Pittsburgh*, April 1994.

[9] M. Harmon, T. Baker, and D. Whalley, "A Retargetable Technique for Predicting Execution Time," *Proceedings of the Real-Time Systems Symposium*, pp.68-77, 1992.

[10] S. Hong and R. Greber, "Compiling Real-Time Programs into Schedulable Code," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and implementation*, pp.166-176, 1993.

[11] D.B. Kirk and J.K. Strosnider, "SMART (Strategic Memory Allocation for Real-Time) Cache Design Using MIPS R3000," *Proceedings of the 11th Real-Time Systems Symposium*, pp.322-330, Orlando, Florida, December 1990.

[12] V. Nirkhe and W. Pugh, "Partial Evaluation of High-Level Imperative Programming Languages with Applications in Hard Real-Time Systems," *Proceedings of the 19th Annual ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, pp.269-280, 1992.

[13] D.M. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No.7, July 1993.

[14] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger, "Developing Real-Time Tasks with Predictable Timing," *IEEE Software*, special issue on Real-time systems, pp.35-44, September 1992.

[15] C. Park and A. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Proceedings of the 11th Real-Time Systems Symposium*, pp.72-81, 1990.

[16] K. Ramamritham, "Allocation and Scheduling of Complex Periodic tasks," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp.108-115, 1990.

[17] M. Spezialetti and R. Gupta, "Perturbation Analysis: A Static Analysis Approach for the Non-Intrusive Monitoring of Parallel Programs," *to appear International Conference on Parallel Processing*, August 1994.

[18] A. D. Stoyenko, "A Real-Time Language With A Schedulability Analyzer," *Ph.D. Thesis*, University of Toronto, August 1987.

[19] H. Tokuda, M. Kotera, and C.W. Mercer, "A Real-Time Monitor for a Distributed Real-Time Operating System," *Technical Report*, Carnegie Mellon University, CMU-CS-88-179, 1988.

[20] J.J.P. Tsai, K-Y. Fang, and H-Y. Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems," *IEEE Computer*, pp.11-23, March 1990.