Priority inversion and its control: An experimental investigation

Article *in* ACM SIGAda Ada Letters · November 1988

DOI: 10.1145/59368.59374

Doug Locke
Locke Consulting LLC

Priority Inversion and Its Control:

An Experimental Investigation

Douglass Locke, IBM FSD
Lui Sha, Ragunathan Rajkumar and John Lehoczky, CMU
Greg Burns, Verdix Corporation

To successfully engineer a large scale real-time system project, we need a disciplined approach to both the logical complexity and the timing complexity inherent in these systems. The logical complexity is managed by the software engineering methodology embodied in Ada while the timing complexity is supported by formal real-time scheduling algorithms [3, 4, 5, 6]. From a software engineering point of view, formal scheduling algorithms translate the complex timing constraints into simple resource utilization constraints. As long as the utilization constraints of CPU, I/O channel and communication media are observed, the deadlines of periodic tasks and the response time requirements of aperiodic tasks will both be met.

There is considerable freedom to modify software provided that the resource utilization remains within specified bounds. Furthermore, should there be a transient overload, the number of tasks missing their deadlines will be a function of the magnitude of the overload and the order in which the tasks miss deadlines is pre-defined [7]. From an implementation point of view, these algorithms belong to the class of static priority algorithms which can be implemented efficiently. Task priorities are functions of the timing constraints, computation requirements and relative importance of tasks. The priorities need not be computed at run-time unless task parameters are modified.

A principal obstacle to the implementation of these formal real-time scheduling algorithms is priority inversion, which was recognized in the previous Ada Real-Time Workshop as a serious problem that must be corrected [1]. Priority inversion is any situation in which a lower priority task holds a resource while a higher priority task is ready to use it. Four changes are needed to bound the waiting time of a higher priority task as a function of the rendezvous durations of shared lower priority servers, and not as a function of the execution times of lower priority tasks.

1. The queue for a task accept must not be managed as a FIFO queue, but rather as a priority queue, ordered according to the priorities of the tasks making the calls. Thus, scheduling of the resulting rendezvous will not result in priority inversion.
2. An unguarded select clause governing a set of task accepts must select first the accept whose queue currently contains the highest priority calling task.
3. A task must always run at the greater of its own priority or the priority of the highest priority task on any of its entry queues. This *priority inheritance* must be transitive in the event that a chained sequence of task entry calls are made.
4. The task priorities must be modifiable at run time in order to respond to changes in the application response time requirements (e.g., due to a system mode change, a sensor must increase its sampling rate).

Of these changes, 1, 2, and 3 were implied by the Steelman requirement that the CPU resource must be scheduled "first-in-first-out within priorities". Change 4 was specifically required by the Steelman scheduling requirement, but omitted in the Ada language definition.

A theoretical investigation [8] has showed that if these changes are made, a higher priority task can be blocked by lower priority tasks at most *min*(m, n) times in a processor, where m is the number of servers that it will share with lower priority tasks and n is the number of lower priority tasks. In the same study, it has been shown that in a uniprocessor environment, if application tasks can be properly structured according to certain rules, then the *priority ceiling protocol*, which adds one additional change to the above list, would lead to freedom from deadlock, and bound the blocking durations by lower priority tasks to at most once.

While these theoretical findings are interesting in their own right, it is imperative that we examine their actual behavior experimentally, because the assumptions made by the theory can only be approximated in practice. To investigate the effects of the intrinsic Ada priority inversion and its control, we conducted a set of experiments with an existing Ada run-time environment which has been modified in accordance with changes 1, 2, and 3 described above. Results of this experimentation using the rate monotonic algorithm are presented in the next section. We believe this subject is of critical importance to all users of Ada for embedded real-time systems, particularly as the existing Ada development tools mature and their inefficiencies, particularly in task and rendezvous management, are removed.

Experimental Results

We now present results from a set of experiments conducted to study the effects of priority inversion in Ada. These results illustrate that the proposed changes lead to considerable improvements in performance.

The experiments were designed as follows. We pick a set of 10 periodic tasks. Each task T_i has its own period P_i and its computational requirements were as follows: T_i has a processing requirement of $C_{i,1}$, after which it has a service requirement of S_i from a server ζ and an additional processing requirement $C_{i,2}$. The server ζ offers a single service and repeatedly services requesting clients for their servicing durations. The periods, the processing and servicing requirements are chosen from a uniform distribution. Each task has a hard deadline defined to be at the end of its current period. All tasks, including the server, were defined in Ada and the computational requirements were represented as execution of tight loops for the duration of the requirements.

The tasks are assigned priorities based on the rate-monotonic priority assignment policy[1], i.e. a task with a shorter period is assigned a higher priority. The server is assigned the lowest priority in the system. Given a task set $\{(C_{1,1}, S_1, C_{1,2}, P_1), ..., (C_{n,1}, S_n, C_{n,2}, P_n)\}$, we systematically scale up the requirements of each task by increasing the value of a weight w, such that the resulting task set $\{(wC_{1,1}, wS_1, wC_{1,2}, P_1), ..., (wC_{n,1}, wS_n, wC_{n,2}, P_n)\}$ just has a task missing its deadline. The utilization at this point is referred to as the *breakdown utilization*. An average of the processor breakdown utilization for a large number of random task sets yields a measure of the performance of the scheduling mechanisms used to schedule these task sets. Higher the utilization, better the performance of the corresponding mechanisms.

We ran the above experiments on three Ada run-time systems. The first was a commercially available Ada run-time system that adheres to the current definition of Ada. This Ada run-time system was then modified to incorporate changes 1, 2 and 3 above. This run-time system is referred to as the run-time system with priority inheritance. The third run-time system incorporated changes 1 and 2 only. That is, entry queues are priority ordered and a *select* statement first selects the accept whose queue currently

[1]The rate-monotonic priority assignment represents the optimal static priority assignment [5].

contains the highest priority calling task. Identical task sets were now scheduled on these three run-time systems. We present the results for 40 random task sets in Table 1 below.

Run-Time System	Average Breakdown	Maximum Utilization	Minimum bUtilization	Maximum Difference w.r.t. PI
w/ Priority Inheritance	85	95	62[2]	0
Priority Ordered	67	89	31	63
Old Ada (as is)	52	84	25	67

Table 1: Performance Comparisons for Single Service

The second column represents the average schedulable utilizations (in percent) achieved by the three run-time systems. The run-time system with priority inheritance achieves about 85% on the average, the run-time system with priority ordered entry queues achieves about 67% while the original run-time system attains only about 52%. Thus, the average performance enhancement provided by the provision of priority inheritance is about 18% w.r.t. the provision of priority ordered queues only and about 33% w.r.t. the old Ada run-time. Columns 3 and 4 provide the maximum and minimum schedulable utilization provided by the three run-time systems on a single task set. The last column gives the maximum enhancement provided by the provision of priority inheritance w.r.t. a single task set. As can be seen, the performance of run-time systems without priority inheritance can span a wide range and the inclusion of priority inheritance can yield immense benefits.

While Table 1 provides the results for random task sets, we repeated the experiment with three task sets characterizing actual real-time systems. The first two task sets represent signal-processing applications while the third task set represents a navigation application. The parameters of these task sets were abstracted to fit the task model for the experiment. The results for these three task sets are provided in Table 2 below. As can be seen, a run-time system with priority inheritance provides a 2:1 enhancement over run-time systems without priority inheritance. The reason behind the high values for the breakdown utilization with priority inheritance is that these task sets are nearly harmonic.

Run-Time System	Average Breakdown	Maximum Utilization	Minimum Utilization	Maximum Difference w.r.t. PI
w/ Priority Inheritance	96	98	95	0
Priority Ordered	49	51	46	48
Old Ada (as is)	46	48	44	51

Table 2: Case Studies with Single Service

We next conducted a third study where instead of the server ζ providing a single service, it provides three services within a *select* statement that is executed within an infinite loop. Each task needs one of these three services and the particular service that is requested by each task was generated from a uniform distribution as well. The results for these studies are presented in Table 3 and are qualitatively similar to

[2]Note that the minimum breakdown utilization is below the Liu and Layland bound [5] of 69%. This is possible because the Liu and Layland bound does not take into account synchronization costs.

the results in Table 1.

Run-Time System	Average Breakdown	Maximum Utilization	Minimum Utilization	Maximum Difference w.r.t. PI
w/ Priority Inheritance	87	94	75	0
Priority Ordered	65	92	31	36
Old Ada (as is)	53	83	25	30

Table 3: Performance Comparisons for Multiple Services

Concluding Remarks

The experimentation has confirmed earlier theoretical analysis that priority inheritance can provide substantial benefits, especially in actual real-time applications where tasks are harmonic or nearly harmonic. However, we are now experimenting with the *priority ceiling protocol*, that guarantees freedom from deadlock, minimizes the blocking of each task to at most once and streamlines the run-time implementation [2, 8].

References

[1] Comhill, D.
Tasking Session Summary.
Proceeding of ACM International Workshop on Real-Time Ada Issues, Ada Newsletter VII, 6, pp 29-32 , 1987.

[2] Goodenough, J. B., and Sha, L.
The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks.
To appear in the Proceedings of the 2nd ACM International Workshop on Real-Time Ada Issues , 1988.

[3] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1 , May, 1986.

[4] Lehoczky, J. P., Sha L and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[5] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[6] Rajkumar, R., Sha, L and Lehoczky, L.
On Countering The Effect of Cycle Stealing in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[7] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
IEEE Real-Time Systems Symposium , 1986.

[8] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report, Department of Computer Science, CMU , 1987.