Technical Report

Linearizability and Schedulability

Björn Andersson

HURRAY-TR-071004
Version: 0
Date: 10-22-2007

Linearizability and Schedulability

Björn Andersson

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

Consider the problem of scheduling a set of tasks on a single processor such that deadlines are met. Assume that tasks may share data and that linearizability, the most common correctness condition for data sharing,must be satisfied.We find that linearizability can severely penalize schedulability. We identify, however, two special cases where linearizability causes no or not too large penalty on schedulability.

Linearizability and Schedulability

Björn Andersson
IPP-HURRAY! Research Group,
Polytechnic Institute of Porto (ISEP-IPP),
Rua Dr. António Bernardino de Almeida 431,
4200-072 Porto, Portugal
bandersson@dei.isep.ipp.pt

Abstract

Consider the problem of scheduling a set of tasks on a single processor such that deadlines are met. Assume that tasks may share data and that linearizability, the most common correctness condition for data sharing, must be satisfied. We find that linearizability can severely penalize schedulability. We identify, however, two special cases where linearizability causes no or not too large penalty on schedulability.

1. Introduction

Consider the problem of preemptive scheduling of n sporadically arriving tasks on a single processor where tasks may share data. A task τ_i is given a unique index in the range $1..n$. A task τ_i generates a (potentially infinite) sequence of jobs. The time when these jobs arrive cannot be controlled by the scheduling algorithm and the time of a job arrival is unknown to the scheduling algorithm before the job arrives.

It is assumed that the time between two consecutive jobs from the same task τ_i is at least T_i. It is assumed that there is a single shared data object in the system. A job may invoke an operation on that data object and the data object will generate a response back to the job (for example, if the shared data object is a linked list then invoking an operation may be requesting to insert an element in the list and the response may be a boolean specifying whether the list was empty before insertion). We assume that *linearizability* [1], the most common correctness criteria for data sharing, must be satisfied. In order to define linearizability we need two other definitions. A history is a sequence of invocations and responses made on an object by a set of tasks. A sequential history is a history in which all invocations have immediate responses. Linearizability is satisfied if every history that can occur to the shared data object is linearizable. A history is linearizable if all these three conditions are true:

C1. invocations and responses of the history can be reordered to yield a sequential history;

C2. that sequential history is correct according to the sequential definition of the object;

C3. if a response preceded an invocation in the original history, it must still precede it in the sequential reordering.

The execution of a job is as follows. A job needs to perform C_i time units in order to finish execution and among that, C_i^{op} of the execution is due to executing an operation on the shared data object. Clearly it holds that $C_i^{op} \leq C_i$. The processor utilization is defined as $U^{processor} = \sum_{i=1}^{n} \frac{C_i}{T_i}$. The utilization of the shared data object is $U^{op} = \sum_{i=1}^{n} \frac{C_i^{op}}{T_i}$. The utilization is defined as $U = \max(U^{processor}, U^{op})$[1].

It is required that the job finishes the execution at most T_i time units after its arrival. In order to satisfy the deadlines and linearizability, a real-time scheduling algorithm S and a data sharing protocol DP is used. We assume that $0 \leq C_i \leq T_i$. Also, recall that we have already mentioned that $C_i^{op} \leq C_i$. It is assumed that C_i, C_i^{op} and T_i are real numbers.

It is common to characterize the performance of scheduling algorithms using the utilization bound. The utilization bound $UB(S, DP)$ of the composition of the scheduling algorithm S and the data sharing protocol DP is the maximum number such that if all tasks are scheduled by S and operations on the shared data object are controlled by DP and if $U \leq UB(S, DP)$ then all tasks meet their deadlines

[1]The careful reader can see that since $C_i^{op} \leq C_i$ it follows that $U^{op} \leq U^{processor}$ and hence the $\max$ function in the expression of U is redundant. We define utilization in this way though because (i) it can be easily extended to multiprocessor systems and (ii) it can be used to show which resource (processor or the shared data object) is the "'bottleneck"'.

and all operations on the shared data object satisfies linearizability. It is interesting to ask whether it is possible to design S and D such that $UB(S, DP) > 0$.

In this paper, we show that for every S and D, it holds that $UB(S, DP)$=0. This low performance is caused by the fact that C3 must be satisfied. As a consequence, we address two special cases which are less penalized by C3 and for those special cases we find that $UB(S, DP) = 1$ and $UB(S, DP) = 0.5$ respectively can be achieved.

The remainder of this paper is organized as follows. Section 2 discusses how linearizability limits schedulability. Section 3 discusses a special case where only read operations and write operations are performed on the shared data object. Section 4 discusses the special case where the time required for each operation is the same. Finally, Section 5 gives conclusions.

2. Linearizability

It is easy to see that for every scheduling algorithm and for every data-sharing protocol, the composition of these causes the utilization bound to be zero because linearizability must be satisfied. Example 1 shows this.

Example 1. *Figure 1 illustrates the task set in the example. Consider two tasks characterized as T_1=1,$C_1 = C_1^{op}$=1/L, T_2=L and $C_2 = C_2^{op}$=1. We choose $L >$2.*

Let us consider an arbitrary job released by τ_2. Let A_2 denote the arrival time of that job. We know that in order to meet deadlines, this job must finish C_2 time units within $[A_2, A_2 + T_2)$. We also know that in order to satisfy linearizability, the response given by the shared data object is equal to some sequential order (according to $C1$). Let SEQ denote this sequential order. Then it holds that this sequential order SEQ must be $< \ldots, OP1', OP2, OP1'', \ldots >$ where $OP1'$ and $OP1''$ denote two different operations by τ_1 and OP2 denotes an operation by τ_2. If we ignore the deadline constraints then it is possible to schedule tasks (and hence the operations) such that such an order exist and hence linearizability would be satisfied.

Let us now reason about linearizability and satisfying deadlines. Consider the sequential order SEQ. Since we do not assume any specific shared data object; the response given by $OP1''$ may depend on $OP2$ and the response by $OP2$ may depend on $OP1'$. This follows from C2. Because of this dependency, it follows that the correct response can only be achieved through scheduling the jobs such that this sequential history is achieved. That is, it must be that a job from τ_1 is executed and then a job from τ_2 is executed non-preemptively and this job from τ_2 must finish its execution before the next job of τ_1 starts executing. Hence it must be possible to perform C_1+C_2 units of execution a time interval of duration T_1.

Using actual numbers gives us that it must be possible to perform $1/L+1$ units of execution during a time interval of duration 1. This is clearly impossible and hence it follows that it is impossible to schedule this task set such that all deadlines are met and linearizability is satisfied.

We can do this reasoning for every choice of L. Letting $L \to \infty$ yields a task set with $U \to 0$ and still a deadline is missed.

3. Only Read/Write operations

The low utilization bound caused by linearizability follows from the fact that we made no assumption on the operations offered by the shared data object. We will now consider a special type of shared data object with only two possible operations: read the entire data from the shared object or write data to the entire shared object.

An efficient shared data object with these operations has been proposed [3]. The main idea is that a vector stores pointers to different versions of the object. When a task invokes a request to write to the object, it finds empty space and writes the new object there. And then, it writes a pointer to this new version in the vector (using the atomic Compare-and-Swap instruction). A task invoking a read operation will obtain a pointer to the last pointer that was inserted in the vector. Observe that writes are performed on new buffers and hence a read operation will never "see" a partially updated object. Also observe that the response of a write operation does not depend on any previous reads and hence it is easy to find a sequential order and consequentially this object is linearizable. The scheme also recycles memory buffers in the vector in order to be able to operate with a finite amount of memory. This scheme (which we call TZ-Buffer) can be used together with EDF and it has $UB(EDF, \mathrm{TZ-Buffer}) = 1$; assuming that the overhead within the data-object (finding an empty buffer, etc.) is zero.

4. Equal execution time for operations

Example 1 in Section 2 showed that the utilization bound can be 0 because linearizability must be satisfied. In the example, the execution times of the tasks were very different. This is reasonable in many real-world applications. For example one task may perform sampling and it inserts the data in a buffer and it has a very small execution time. Another task reads data from the buffer, performs processing and outputs the result on a display and it has a very large execution time.

Despite the fact that execution times of tasks may be very different; it is often the case that the operations on shared data objects are small and the execution times are approximately the same. Consider for example a FIFO queue; the

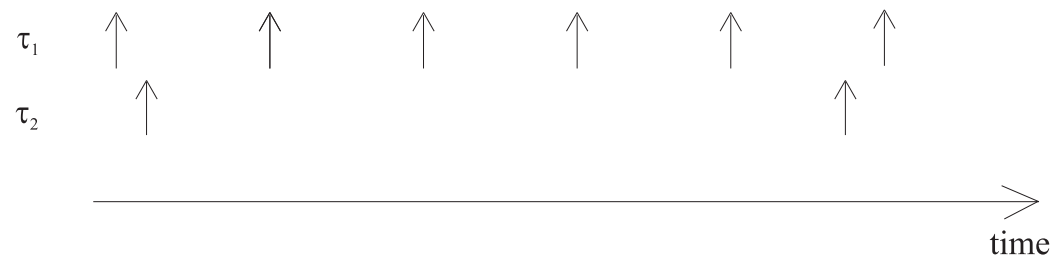

Figure 1. An example of a task set with utilization arbitrary close to zero and it misses a deadline because linearizability must be satisfied. An arrow pointing upwards indicates the arrival time of a job.

time required to perform insert and remove is approximately the same. Also, consider a hierarchical data structure (as a tree). Searching, adding and deleting typically have time complexities $O(\log k)$, where k is the number of elements, and hence the difference in execution times cannot be too high assuming that there is an upper bound on the number of elements. It has also been observed that in practice many operations on shared data objects have a short execution time [4, 5].

For these reasons, we will now consider the case where $\forall i, j : C_i^{op} = C_j^{op}$. We will consider Earliest-Deadline-First (EDF) scheduling [2]. At time t, it assigns priorities to jobs that have arrived at time t but have not yet finished execution. Let $A(J_i)$ denote the arrival time of a job J_i. Then the priority of job J_i is higher than the priority of job J_j if $A(J_i) + T_i < A(J_j) + T_j$. We will consider the non-preemptive critical-section (NPCS) protocol. It means that all operations on the shared data object are protected by a critical section and when a task executes in the critical section, it executes non-preemptively; when the task leaves the critical section, the task can be preempted again.

The performance of EDF with the NPCS protocol is provided by Theorem 1.

Theorem 1. *Consider a task set such that* $\forall i, j : C_i^{op} = C_j^{op}$ *and* $\sum_{j=1}^{n} C_j/T_j \leq 1/2$ *and this task set is scheduled by EDF and the NPCS protocol. We claim that this task set meets all deadlines.*

Proof. The proof is by contradiction. Suppose that the theorem was incorrect. Then there must exist a task set such that:

$$\forall i, j : C_i^{op} = C_j^{op} \tag{1}$$

and

$$\sum_{j=1}^{n} C_j/T_j \leq 1/2 \tag{2}$$

and the task set is scheduled by EDF and the NPCS protocol and at least one deadline was missed.

Let us consider the earliest time when a deadline was missed. M denotes one of those jobs. A_i denotes the arrival time of M. τ_i denotes the task that released M. Let us consider two cases:

1. $C_i/T_i > 0.5$

   This contradicts Equation 2.

2. $C_i/T_i \leq 0.5$

   Let us reason about this case. t_1 denotes the deadline of M and t_0 denotes the latest time such that $t_0 < t_1$ and the processor is busy during $[t_0, t_1)$ and the processor is idle just before t_0. For convenience we let L denote $t_1 - t_0$. We know that $[t_0, t_1)$ must be non-empty since $C_i > 0$ and $T_i > 0$. Let us define two classes of tasks:

   $$\tau^{classA,i} = \{\tau_j : T_j < T_i\} \tag{3}$$

   end

   $$\tau^{classB,i} = \{\tau_j : (T_j \geq T_i) \wedge (j \neq i)\} \tag{4}$$

   Let us consider a task τ_j in $\tau^{classA,i}$. If τ_j arrived after time $t_1 - T_j$ then its deadline is later than t_1 and hence it will have a lower priority than M. Also observe that such a job from task τ_j arrives after A_i because $T_j < T_i$. Hence this job cannot cause any delay to the job M. Consequently, we delete this job released by τ_j and then M still misses its deadline. For this reason we obtain that the amount of work performed by task τ_j in $\tau^{classA,i}$ during $[t_0, t_1)$ is at most:

   $$\max(\lfloor \frac{L - T_j}{T_j} \rfloor + 1, 0) \cdot C_j \tag{5}$$

   Let us now consider a task τ_j in $\tau^{classB,i}$. If none of these jobs executed just after time A_i then we obtain that the amount of work performed by task τ_j in $\tau^{classB,i}$ during $[t_0, t_1)$ is at most:

$$\max(\lfloor\frac{L-T_j}{T_j}\rfloor+1,0)\cdot C_j \quad (6)$$

Let us now consider a task τ_j in $\tau^{classB,i}$. If one of these jobs executed just after time A_i then let τ_j denote this task. Observe that this task may release a job with a lower priority than M but it will still execute in $[t_0,t_1)$. The amount of execution it will perform is C_j^{op}. With reasoning similar to Equation 5-6 we obtain that the amount of work performed by this task τ_j in $\tau^{classB,i}$ during $[t_0,t_1)$ is at most:

$$C_j^{op}+\max(\lfloor\frac{L-T_j}{T_j}\rfloor+1,0)\cdot C_j \quad (7)$$

It is also clear that for task τ_i we have that the amount of work performed by task τ_i during $[t_0,t_1)$ is at most:

$$\max(\lfloor\frac{L-T_i}{T_i}\rfloor+1,0)\cdot C_i \quad (8)$$

Taking Equation 5-8 together and simplifying yields that the amount of work performed by all tasks during $[t_0,t_1)$ is at most:

$$C_j^{op}+\sum_{j=1}^{n}\max(\lfloor\frac{L-T_i}{T_i}\rfloor+1,0)\cdot C_i \quad (9)$$

We know that $C_j^{op}=C_i^{op}$ and $C_i^{op}\leq C_i$ and that $C_i\leq T_i/2$. Using this on Equation 9 yields that the amount of work performed by all tasks during $[t_0,t_1)$ is at most:

$$\frac{T_i}{2}+\sum_{j=1}^{n}\max(\lfloor\frac{L-T_i}{T_i}\rfloor+1,0)\cdot C_i \quad (10)$$

Simplifying yields that the amount of work performed by all tasks during $[t_0,t_1)$ is at most:

$$\frac{T_i}{2}+\sum_{j=1}^{n}\lfloor\frac{L}{T_i}\rfloor\cdot C_i \quad (11)$$

Since a deadline is missed it must be that:

$$\frac{T_i}{2}+\sum_{j=1}^{n}\lfloor\frac{L}{T_i}\rfloor\cdot C_i > L \quad (12)$$

Since τ_i was the task that missed a deadline, it follows that $T_i\leq L$. Using it on Equation 12 and rewriting yields:

$$\frac{L}{2}+\sum_{j=1}^{n}\lfloor\frac{L}{T_i}\rfloor\cdot C_i > L \quad (13)$$

Simplifying Equation 13 yields:

$$\sum_{j=1}^{n}\lfloor\frac{L}{T_i}\rfloor\cdot C_i > \frac{L}{2} \quad (14)$$

Dividing by L and relaxing yields:

$$\sum_{j=1}^{n}\frac{C_i}{T_i} > \frac{1}{2} \quad (15)$$

But this contradicts Equation 2.

We can see that regardless of the case we obtain a contradiction. And hence the theorem is true. □

5. Conclusions

We have seen that satisfying linearizability causes the utilization bound to drop to 0. We also saw two special cases where the utilization is greater than zero.

It deserves to find out which data objects can be shared while attaining a non-zero utilization bound. For those data objects where the utilization bound necessarily is zero it deserves to find (i) limitations on the task sets for which the utilization bound is non-zero and (ii) whether alternative correctness criteria can be used.

Acknowledgements

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para Ciência e Tecnologia - FCT) and the ARTIST2 Network of Excellence on Embedded Systems Design.

References

[1] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, May 1990.

[2] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for the Computing Machinery*, 20:46–61, 1973.

[3] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proc. of Real-Time Computing Systems and Applications*, pages 247–254, Hong Kong, China, Dec. 13–15, 1999.

[4] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proc. of ACM SIGMETRICS Performance Evaluation*, pages 320–321, Massachusetts, USA, June 17–20, 2001.

[5] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proc. of the 3rd international workshop on Software and performance*, pages 55–67, Rome, Italy, July 24–26, 2002.