

# Response time analysis of software transactional memory-based distributed real-time systems

Sherif F. Fahmy  
ECE Dept., Virginia Tech  
Blacksburg, VA 24061, USA  
fahmy@vt.edu

Binoy Ravindran  
ECE Dept., Virginia Tech  
Blacksburg, VA 24061, USA  
binoy@vt.edu

E. D. Jensen  
The MITRE Corporation  
Bedford, MA 01730, USA  
jensen@mitre.org

## ABSTRACT

We consider distributed real-time systems where concurrency control is managed using software transactional memory (or STM). For such a method we propose an algorithm to compute an upper bound on the response time. The proposed algorithm can be used to study the behavior of systems where node crash failures are possible. We compare the result of the proposed algorithm to a simulation of the system being studied in order to determine its efficacy. The results of our study indicate that it is possible to provide timeliness guarantees for systems programmed using STM.

## Categories and Subject Descriptors

C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

## Keywords

Software transactional memory, response time, real-time

## 1. INTRODUCTION

Recently, there has been a shift in the computer industry from increasing clock rates to designing multicore and hyper-threading architectures in the quest to produce faster computers [22]. This paradigm shift means that programmers can no longer depend primarily on faster processors to increase the speed of their programs. Improving software performance now depends on writing, correct, concurrent code. Unfortunately, the traditional method of using locks and condition variables places a heavy burden on programmers—one that, many people believe, they are not very well suited for [13].

The difficulty of reasoning about lock-based concurrency control can cause a large number of subtle program errors. Among the more common errors encountered are deadlocks, livelocks, lock convoying, and, in systems where priority is important (e.g. real-time systems), priority inversion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

Copyright 2009 ACM 978-1-60558-166-8/09/03 ...\$5.00.

Thus, alternatives to lock-based concurrency control have become the subject of intense study in the literature. A particularly promising alternative to lock-based concurrency control (for non-I/O code) is transactional memory. There have been significant recent efforts to apply the concepts of transactions to shared memory. Such an attempt originated as a purely hardware solution [8, 12] and was later extended to deal with systems where transactional support was migrated to the software domain [20].

While STM has many promising features, it is not a silver bullet. Some problems associated with STM include handling irrevocable instructions such as I/O, the weak atomic semantics of some of the current implementations [6] and the overhead of retries. Despite these disadvantages, its semantic simplicity makes it a very promising alternative to lock-based concurrency control.

In this paper, we present an algorithm for computing a worst-case bound on the response time of tasks in a real-time distributed system, where failures may occur and concurrency control is managed using STM.

## 2. PREVIOUS WORK

Since the seminal papers about hardware and software transactional memory were published, renewed interest in the field has resulted in a large body of literature on the topic (e.g. see [4]). This body of work encompasses both purely software transactional memory systems, and hybrid systems where software and hardware support for transactional memory are used together to improve performance. Despite this large body of work, to the best of our knowledge, very few papers investigate STM for distributed systems [5, 9, 14].

There has also been a dearth of work on real-time STM systems. Notable work on transactional memory and lock-free data structures in real-time systems include [1–3, 10, 15]. However, most of these works only consider uni-processor systems (with [10] being a notable exception). In [15], the authors consider a single processor system with priority scheduling. While in [1–3], the schedulability analysis is utilization based and only considers a single processor. In [10], a multiprocessor machine is considered but the scheduling is restricted to quanta-based scheduling algorithms such as the pfair algorithm.

In this paper, we consider using STM as the concurrency control mechanism for (non-I/O code in) distributed real-time systems. Toward this, we propose a method for computing an upper bound on the worst-case response time of periodic tasks, programmed using STM, running in a dis-

tributed real-time system that employs Earliest Deadline First (EDF) scheduling.

### 3. GENERAL FRAMEWORK

There are different ways of incorporating STM into distributed systems. In the framework we consider, there are logical entities called “transactions” that consist of several sub-transactions. Each sub-transaction executes within one processor and invokes its successor using an RPC when it has completed execution. This general framework can be further divided into several different approaches. In a position paper [6], the authors discuss three different approaches for incorporating STM into this framework. First, the authors consider a system where distributed shared blocks can occur (i.e., where a sub-transaction enters a critical section and then makes a remote invocation without exiting the critical section). This results in a distributed transaction that needs a distributed commit protocol to ensure atomicity. Another approach considered in [6] is to design a distributed cache coherence protocol and, thus, present the abstraction of shared memory in the distributed system on top of which STM can be designed. The third, and last, approach considered in [6] is a cross between the first two, where code and data mobility are both allowed in order to reduce the cost of communication delay.

In this paper, we consider a system of transactions programmed on a distributed system. Each transaction consists of several sub-transactions executing on single nodes that are subject to crash failures. Concurrency control on each node is managed using STM. A sub-transaction makes an invocation (a procedure call or an RPC, depending on whether the successor resides on the same node) after it has finished execution. We do not allow cross-node critical sections. In the rest of this paper, we use the term task instead of sub-transaction in order to follow the terminology used in the rest of the literature, e.g., [18, 19].

We present a method for analyzing the worst-case response times in such a system. In order to do this, we first show how to extend Spuri’s response time analysis technique [21] to include the overhead associated with the retry behavior of STM on a single machine without considering offsets in Section 4.2. We then extend this analysis to include offsets in Section 5.2. Including offsets is necessary since it enables us to handle the precedence constraints of a distributed system where tasks make RPC calls to remote nodes and therefore some tasks cannot start before their predecessor makes an invocation.

The ability to use offsets and jitters to represent these precedence constraints is discussed in Section 7. In Section 6.1, we show how failures can be considered in the analysis by ensuring that exception handlers can be executed if necessary. Our goal is to prove that it is possible to provide real-time guarantees for distributed systems where concurrency control is managed using STM. Introducing STM to the repertoire of programming tools available for the real-time programmer can considerably improve the quality of distributed concurrent real-time programs and reduce the software development time by reducing the complexity of the programming environment. As such, this paper is our first step towards achieving the goal of studying STM for distributed real-time systems in all its possible varieties as outlined in [6].

## 4. TASKS WITH JITTER

Given a set of periodic, independent tasks scheduled by EDF on a single processor, Spuri, in [21], proposed an algorithm for computing an upper bound on the worst-case response time for a task. In this section we extend the algorithm to consider tasks with mutually exclusive resource access requirements that are programmed using software transactional memory. We extend Spuri’s analysis to consider these tasks and compare the tightness of the bound we obtain by comparing it to the utilization based schedulability analysis of lock-free code proposed in [2].

Spuri’s idea for computing an upper bound on worst-case response time is based on finding a “critical instant”, the release time of a task,  $\tau_a$ , that would cause it to experience maximal interference from other tasks. This critical instant is found in a busy period, which is defined as a period of time during which a processor is busy executing tasks in the system. The following theorem is helpful in finding that critical instant:

**THEOREM 1** (SPURI [21]). *The worst-case response time of a task  $\tau_a$  is found in a busy period in which all other tasks are released simultaneously at the beginning of the busy period, after having experienced their maximum jitter.*

The proof of this theorem, presented in [21], rests on the fact that the conditions in Theorem 1 result in the largest number of interferences from other tasks. The same condition holds for tasks using STM, since the largest number of interferences will result in the largest number of transaction retries and hence the worst-case response time. Note that we make the assumption that each interference by a task results in a transactional retry. This is a pessimistic assumption since the task may not be executing its transactional code at this point in time, or the transactional operations may not conflict before the transaction commits. However, we make this assumption to give our analysis the nature of an upper bound. In Section 4.2, we present the modified analysis for independent tasks on a single processor.

### 4.1 Task model

We consider a task model with periodic tasks scheduled with the EDF discipline. In this model, the system is composed of a set of  $N$  periodic tasks executing in a single processor. Each task  $\tau_i$  is activated periodically with a period of  $T_i$ , has a computation time of  $C_i + m_i s$ , where  $C_i$  is the computation time of the task instance without considering the transactional part of the code,  $m_i$  is the number of times transactional code is executed in the task activation and  $s$  is the computation cost of the transactional part of the code. In the rest of the paper, we shall refer to a task activation as *job*. Each job has a relative deadline  $d_i$  and a release jitter bounded by  $J_i$ . We refer to the absolute deadline of a job as  $D_i$ .

### 4.2 Analysis

In this section, we compute the worst-case contribution of a task  $\tau_i$  to the response time of the task under analysis  $\tau_a$ . Specifically, we compute the worst-case contribution of task  $\tau_i$  during a busy period of duration  $t$  when the deadline of  $\tau_a$  is  $D$ . Without loss of generality, we label the time at which the busy period starts as  $t_0$ , and measure the duration of the busy period,  $t$ , and the deadline of  $\tau_a$ ,  $D$ , from  $t_0$ .

As per Theorem 1, the worst-case contribution of a task,  $\tau_i$ , occurs when it is released at the start of the busy period after having experienced its maximum jitter. This scenario is chosen so as to maximize the number of instances of  $\tau_i$  that occur during the busy period in order to maximize interference.

Only jobs with a deadline less than or equal to  $D$  can interfere with  $\tau_a$ , also, jobs that start outside the busy period,  $t$ , do not contribute to the interferences that occur within that period. Using this information, we can compute the maximum number of jobs of  $\tau_i$  that can interfere with  $\tau_a$ .

From [17], we know that the number of jobs of  $\tau_i$  within the busy period,  $p_t$ , is:

$$p_t = \left\lceil \frac{t + J_i}{T_i} \right\rceil \quad (1)$$

and the number of task instances with deadlines at or before  $D$  is:

$$p_D = \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \quad (2)$$

Since both conditions,  $p_D$  and  $p_t$ , must be satisfied, the maximum number of interferences of the jobs of  $\tau_i$  in  $\tau_a$  is:

$$n_i = \min \left( \left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \quad (3)$$

The zero that appears as a subscript in the equation above causes the result of the bracketed expression to be zero if its value is negative. This is necessary because if  $d_i > D$ , no activation of  $\tau_i$  will interfere with  $\tau_a$ . Thus, the worst-case contribution of  $\tau_i$  to the response time of  $\tau_a$  is:

$$W_i(t, D) = n_i(C_i + m_i s) \quad (4)$$

Given this equation, it is possible to compute the response time of  $\tau_a$  after having determined the critical instant. Unfortunately, we do not know which instant in the busy period is the critical instant. However, it is known that the critical instant can be found either at the beginning of the busy period, or at an instant of time such that the deadline of the analyzed job of  $\tau_a$  coincides with the deadline of a task  $\tau_i$ 's job. Otherwise, it would be possible to make the activation time of  $\tau_a$  earlier, without changing the schedule, to increase the response time. The set of instants,  $\Psi$ , at which the deadline of  $\tau_a$ 's job coincides with the deadline of the job of some other task in the busy period, is:

$$\Psi = \bigcup \{(p-1)T_i - J_i + d_i\} \quad (5)$$

$$\forall p = 1 \dots \left\lceil \frac{L + J_i}{T_i} \right\rceil, \forall i$$

In the above equation,  $L$  is the worst-case length of a busy period. The following recurrence relation can be used in computing  $L$ :

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil C_i^{exe} \quad (6)$$

where  $C_i^{exe}$  is an estimate of the processing load placed on the processor by a job during the busy period.

Equation (6), one of the many recurrence equations that can be found in response time analysis [11], can be solved by starting with a small initial value for  $L$  and then iterating until the equation converges. The equation is guaranteed to converge if the system is not over-utilized (i.e., the load is under 100%). The execution time of a job, without considering any interference, is  $C_i + m_i s$ . To this execution time, we need to add the load that a job can place on the processor during the busy period if it interferes with another job. For uni-processor systems scheduled using EDF an interference can only occur when a new job arrives with a lower deadline than those already executing causing a context switch. Thus, a job can cause, at most, one retry in some other job. Therefore, the load a job places on the processor during the busy period is  $C_i + m_i s + s$ , giving us:

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil (C_i + m_i s + s) \quad (7)$$

Thus, we can compute the set of critical instants by subtracting  $d_a$  from each element in  $\Psi$ . We then consider all the critical instants in our analysis to find the critical instant that gives us the worst-case response time.

There may be several jobs of  $\tau_a$  in the busy period, therefore we need to examine all of these jobs in order to determine which one of them results in the worst-case response time. Assuming that the first instance of  $\tau_a$  occurs  $A$  time units after the start of the busy period, the completion time of job  $p$  of  $\tau_a$ ,  $w_a^A(p)$ , can be computed as:

$$w_a^A(p) = p(C_a + m_a s) + \sum_{\forall i \neq a} W_i(w_a^A(p), D^A(p)) + I s \quad (8)$$

where  $I$  is the maximum number of interferences that can occur in jobs of higher priority (lower deadline) than the instant of  $\tau_a$  being studied as expressed in Equation (9).

$$I = \sum_{\forall i} \min \left( \left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \quad (9)$$

In Equation (8), the term  $D^A(p)$  is the deadline of job  $p$  when the first job of  $\tau_a$  occurs  $A$  time units after the start of the busy period and can be computed as:

$$D^A(p) = A - J_a + (p-1)T_a + d_a \quad (10)$$

The response time is obtained by subtracting the activation time (start time) from the completion time of each task:

$$R_a^A(p) = w_a^A(p) - A + J_a - (p-1)T_a \quad (11)$$

Next we need to determine the set of values we will use for  $A$ . Note that we have already established that the critical instant can only be in the set  $\Psi$  computed in Equation (5). We now further narrow down the set of values that need to be considered. Naturally, for each value of  $p$  we only need to check values of  $A$  within one period. Thus, we can further narrow down the critical instants we should consider to:

$$\Psi^* = \{\Psi_x \in \Psi \mid (p-1)T_a - J_a + d_a \leq \Psi_x < pT_a - J_a + d_a\} \quad (12)$$

For each of the values,  $\Psi_x$ , in  $\Psi^*$ , we need to check the values  $A(\Psi_x) = \Psi_x - [(p-1)T_a - J_a + d_a]$ .

Finally, we determine the worst-case response time by examining all instants of  $\tau_a$  within the busy period and taking the maximum response time as our result:

$$R_a = \max R_a^A(p) \quad (13)$$

$$\forall p = 1 \dots \left\lceil \frac{L-J_a}{T_a} \right\rceil, \forall A(\Psi_x) \mid \Psi_x \in \Psi^*$$

### 4.3 Experimental Evaluation

In this section, we experimentally evaluate the response time analysis we obtain by comparing it to the utilization bound schedulability analysis for lock-free code obtained in [2]. Before we do that, we restate the main result of [2] in terms of the notations used in our analysis. The utilization bound for EDF scheduled periodic lock-free code can be expressed as follows:

$$\sum_{\forall i} \frac{C_i + m_i s + s}{T_i} \leq 1 \quad (14)$$

The term  $m_i s$  is not present in the analysis in [2]. This occurs because we separate the execution time of the code without the transactional component,  $C_i$  and the cost of the transactional component,  $s$ . No such distinction occurs in [2] and hence our  $C_i + m_i s$  is equivalent to their  $C_i$ .

In order to compare the proposed technique to the utilization method developed in [2], we performed a series of experiments. We considered a system where the number of tasks, the execution time of the transactional part of the code,  $s$ , the execution times of the non-transactional part of the code,  $C_i$ , and the number of times that transactional code is executed,  $m_i$ , are all randomly determined. The period of each task is varied to obtain different utilizations and jitters are set to zero since the analysis in [2] does not consider jitter.

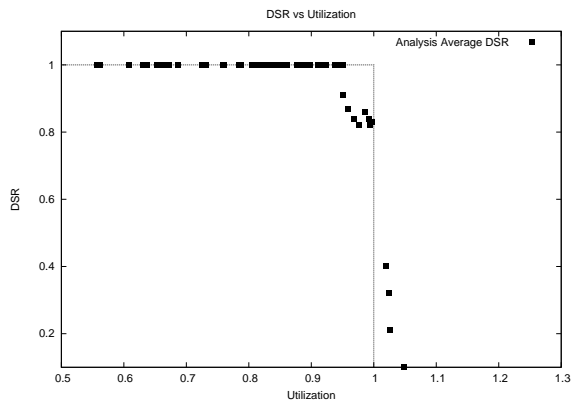


Figure 1: DSR vs. Utilization

We recorded the average Deadline Satisfaction Ratio (DSR), defined as the ratio of the number of jobs that meet their deadlines to the total number of jobs, against the utilization of the system. We used Equation (14) in computing the utilization. Thus, the utilization bound method provides scheduling guarantees for all utilizations under one (this is depicted in Figure 1 as a bounding box).

As can be seen, the response time analysis is equivalent to the utilization bound analysis up to utilization factors

close to one. However, the proposed analysis begins to miss deadlines before the utilization reaches one. Since both the analysis in [2] and the proposed analysis assume that each job interference causes a retry, their pessimism should be about the same. We believe the difference in performance occurs, despite the similarity of the pessimism of both algorithms, because in Equation (7) we extend the busy period to include the cost of retries. Then in Equation (5) we use this extended busy period to compute the number of jobs to consider in the busy period without considering that some of this time will be used for the retries rather than job executions. Thus, we may introduce more jobs into the analysis than necessary, however, this only manifests itself for systems where the overhead introduced by Equation (7) is very large. For most cases, the proposed algorithm provides schedulability up-to the 100% utilization point.

## 5. TASKS WITH JITTER AND OFFSETS

In this section, we study a system where groups of tasks are grouped together into logical entities which we shall call “transactions”. Each transaction is activated by a periodic external event. Once this external event has arrived, the tasks in each transaction begin execution after a certain time period, which we refer to as the offset, has passed. This model can be used to model distributed systems. Already, several papers have been published showing how to obtain an upper bound on the response time of distributed systems programmed using this task model, e.g., [7, 17, 19]. Those attempts are based on the Holistic analysis first proposed by Tindell and Clark [23]. In this section, we extend such analysis to deal with STM based concurrency control management.

### 5.1 Task model

We consider a system composed of a set of tasks executing on the same processor. These tasks are grouped into logical entities referred to as transactions. Each transaction,  $\Gamma_i$ , is activated by a periodic external event with period  $T_i$ . A transaction consists of  $n_i$  tasks (not to be confused with the  $n_i$  used in the analysis of Section 4.2). We designate the tasks  $\tau_{ij}$ , with the first subscript,  $i$ , identifying the transaction the task belongs to, and the second subscript,  $j$ , specifying the order of the task within the transaction in non-decreasing order of offsets.

Each of these tasks has an execution time of  $C_{ij} + m_{ij}s$ , where  $C_{ij}$  and  $s$  are the execution times of the transactional and non-transactional part of a task, respectively, and  $m_{ij}$  is the number of times transactional code is invoked in a task. Tasks are activated after a certain time, which we shall refer to as the offset, elapses from the arrival of the external event that triggered the transaction. For each  $\tau_{ij}$ , its offset is designated  $\phi_{ij}$ . We also allow a task to suffer release jitter which is bounded by the term  $J_{ij}$ . Both offsets and jitter can be larger than the period of their transaction.

### 5.2 Analysis

In this section, we compute the worst-case response time for task  $\tau_{ab}$ . In order to do this, we must determine the worst-case contribution of each transaction to the response time of the task under analysis. The theorem below can be used for this purpose:

**THEOREM 2** (PALENCIA AND HARBOUR [17]). *The worst-*

case contribution of transaction  $\Gamma_i$  to the response time of a task  $\tau_{ab}$  is obtained when the first activation of some task  $\tau_{ik}$  that occurs within the busy period coincides with the beginning of the busy period, after having experienced the maximum possible delay, i.e., the maximum jitter,  $J_{ik}$ .

Again, this theorem is based on the fact that the worst-case contribution will occur when the most number of tasks are released within the busy period. For the sake of being concise, we will not reproduce the whole derivation of the analysis, which can be found in [17], but will only include the final results and the modifications necessary for accommodating STM.

The worst-case contribution of a task,  $\tau_{ij}$ , to the response time of the task under analysis,  $\tau_{ab}$ , during a busy period of duration  $t$  and deadline  $D$ , when the task whose activation time coincides with the start of the busy period is  $\tau_{ik}$ , is:

$$W_{ijk}(t, D) = \left( \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \min \left( \left\lfloor \frac{t - \varphi_{ijk}}{T_i} \right\rfloor, \left\lfloor \frac{D - \varphi_{ijk} - d_{ij}}{T_i} + 1 \right\rfloor \right) \right)_0^{(C_{ij} + S)} \quad (15)$$

where  $\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) \bmod T_i$ . Thus, the contribution of transaction  $\Gamma_i$  is the summation of the contribution of all its tasks:

$$W_{ik}(t, D) = \sum W_{ijk}, \quad \forall j \in \Gamma_i \quad (16)$$

In order to make the analysis tractable, the worst-case contribution of a transaction,  $\Gamma_i$ , is considered to be the maximum of all possible contributions that could have been caused by considering each of the tasks of  $\Gamma_i$  as the start of the busy period:

$$W_i^*(t, D) = \max(W_{ik}(t, D)), \quad \forall k \in \Gamma_i \quad (17)$$

We number the activations of a job within the busy period using the index  $p$ , and consider the first activation to start within the busy period to have an index of  $p = 1$ . The activations that start before the busy period and suffer their maximum jitter to start at the beginning of the busy period have indices  $p \leq 0$ . Thus, we can determine the index of the first,  $P_{0,ijk}$ , and last,  $P_{L,ijk}$ , activations to contribute to the busy period as follows:

$$P_{0,ijk} = - \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + 1 \quad (18)$$

and

$$P_{L,ijk} = \left\lfloor \frac{L - \varphi_{ijk}}{T_i} \right\rfloor \quad (19)$$

where  $L$  is the maximum length of the busy period as computed in Equation (7).

Thus, like in Section 4.2, the set of values to analyze is:

$$\Psi = \bigcup \{ \varphi_{ijk} + (p-1)T_i + d_{ij} \} \quad (20)$$

$$\forall p = P_{0,ijk} \cdots P_{L,ijk}, \quad \forall j, k \in \Gamma_i$$

Thus, if the first activation of  $\tau_{ab}$  occurs after  $A$  time units from the start of the busy period, the worst-case completion time of activation  $p$  of task  $\tau_{ab}$  can be computed as:

$$W_{abc}^A(p) = (p - P_{0,ijk} + 1)(C_{ab} + s) \quad (21)$$

$$+ W_{ac}^-(W_{abc}^A(p), D_{abc}^A(p)) + \sum_{\forall i \neq a} W_i(W_{abc}^A(p), D_{abc}^A(p))$$

where  $W_{ac}^-$  is the result of Equation (16) without considering the contribution of  $\tau_{ab}$  and  $D_{abc}^A(p)$  is the deadline of activation  $p$  when the first one occurs at time  $A$ :

$$D_{abc}^A(p) = A + \varphi_{abc} + (p-1)T_a + d_{ab} \quad (22)$$

Thus, we can obtain the response time of a task by subtracting from the completion time the arrival time of the external event:

$$R_{abc}^A(p) = W_{abc}^A(p) - A - \varphi_{abc} - (p-1)T_a + \phi_{ab} \quad (23)$$

Also, as in Section 4.2, we only need to check the value of  $A$  within one period, thus, the points we need to check are:

$$\Psi^* = \{ \Psi_x \in \Psi \mid \varphi_{abc} + (p-1)T_a + d_{ab} \leq \Psi_x < \varphi_{abc} + pT_a + d_{ab} \} \quad (24)$$

For each value of  $\Psi_x$  above, we check  $A = \Psi_x - [\varphi_{ijk} + (p-1)T_a + d_{ab}]$ . Naturally, the worst-case response time is the maximum response time obtained from the analysis, i.e.,

$$R_{ab} = \max(R_{abc}^A(p)) \quad (25)$$

$$\forall p = P_{0,abc} \cdots P_{L,abc} \quad \forall c \in \Gamma_A, \quad \forall A \in \Psi^*$$

## 6. HANDLING FAILURES

In this section, we extend our analysis to take failures into account. In some distributed systems, failures are the norm rather than the exception. Therefore, it is necessary to provide some form of guarantee on system performance in their presence. We assume that each task,  $\tau_{ij}$ , has an exception handler that can be used to restore the system to a safe state in case of failure, and that this exception handler has an execution time  $C_{ij}^h$  and relative deadline  $d_{ij}^h$ . The absolute deadline of the handler is relative to the time that failure is detected,  $t_f$ , i.e.,  $D_{ij}^h = t_f + d_{ij}^h$ .

When a node fails, all the jobs executing on that node cease to exist. Since we are considering “transactions” where a sequence of consecutive jobs execute within one logical computational context, it is necessary to understand the effect of failures on this abstraction. Naturally, a failure may fragment a transaction leading to several orphan jobs (i.e., jobs that have been disconnected from their downstream predecessor due to node failure). These jobs need to be identified and their exception handlers need to be executed in order to restore the system to a safe state.

Therefore, in order to have a fault-tolerant system, it must be possible to execute the exception handlers before their deadlines when failure occurs. In this section, we show how we can take the execution time of the exception handlers into account when computing response times in order to ensure safe execution of the system in the presence of failures.

### 6.1 Analysis

We need to determine the maximum number of exception handlers that can execute within a busy period in order to take their overhead into account. As in Sections 4.2 and 5.2, there are two conditions that determine the number of jobs,

in this case exception handlers, that can contribute to the worst-case response time of an instance of a task,  $\tau_{ab}$ , within busy period of duration  $t$ ; 1) The number of exception handlers that execute within the duration of the busy period (including exception handlers that were released before the start of the busy period but whose activation time is delayed until the start of the busy period), and 2) the number of handlers with deadline less than or equal to the deadline of the job being analyzed.

From [17], we know that the number of interferences from the jobs of task  $\tau_{ij}$  that occur from jobs that start at the beginning of a busy period, after suffering some jitter, is:

$$x_i = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \quad (26)$$

Each of these activations has an associated exception handler. In the worst-case, each job executes to completion, thus, using up as much processor time as possible, and then an error occurs that causes the trigger of its exception handler, thus, using up more processor time to execute the handler. It is this worst-case scenario from which we derive the deadline of the exception handlers. Note that since we are considering activations of the same task, and all of these activations start at the beginning of the busy period, their exception handlers have the same deadline which is:

$$D_{ij}^{h_{first}} = d_{ij} + d_{ij}^h \quad (27)$$

If we consider, without loss of generality, that the beginning of the busy period  $t_B$  is time zero, the contribution of these exception handlers is  $x_i C_{ij}^h$  if  $D_{ij}^{h_{first}} \leq D$  and zero otherwise.

We now turn our attention to determining the number of interferences that occur from job instances started within the busy period. Below, is the equation that determines the number of instances that can occur within a busy period of duration  $t$ :

$$n_{inst} = \left\lfloor \frac{t - \varphi_{ijk}}{T_i} \right\rfloor \quad (28)$$

For each of these activations, the deadline of their handler can be computed as:

$$De = \varphi_{ijk} + (p-1)T_i + d_{ij} + d_{ij}^h \quad (29)$$

$$\forall p = 1 \dots n_{inst}$$

and they each contribute a factor of  $C_{ij}^h$  to the worst-case response time if their deadline is less than or equal to  $D$ .

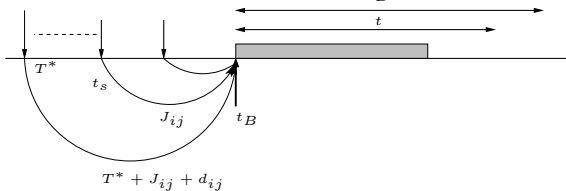


Figure 2: Scenario for calculating worst-case contribution

At this point we have computed the contribution of the execution time of handlers of the activations of  $\tau_{ij}$  that start at or after the beginning of the busy period  $t_B$ . Figure 2 depicts the types of scenarios we will be considering. The term

in Equation (26) represents the number of jobs that can be delayed at most  $J_{ij}$  so that their activation starts at  $t_B$ . In Figure 2, the first such job starts at  $t_s$ ; other jobs that follow it will be delayed an amount of time less than  $J_{ij}$  in order to start at the beginning of the busy period. Equation (28) computes the number of jobs that will start after the beginning of the busy period, i.e., after  $t_B$ . However, if failures are not considered, jobs that start before  $t_s$ , such as the one depicted as starting at time  $T^*$  in Figure 2, do not contribute to the busy period. This lack of contribution occurs because even if these jobs were delayed  $J_{ij}$ , their activation time would fall before  $t_s$ . However, now that we consider failures, it is possible for these tasks to contribute to the worst-case response time if their exception handlers start within the busy period. Using similar reasoning as in Section 5.2, we compute the worst-case contribution of these exception handlers by considering how many can start at the beginning of the busy period. The latest start time of a handler whose job starts at  $T^*$  is:

$$S = T^* + d_{ij} \quad (30)$$

If this start time,  $S$ , is greater than or equal to  $t_B$  then the handler will contribute to the busy period. In other words:

$$T^* + d_{ij} \geq T^* + nT_i + J_{ij} \quad (31)$$

which gives us:

$$n < \frac{d_{ij} - J_{ij}}{T_i} \quad (32)$$

Since  $n$  is an integer, Equation (32) resolves to:

$$n = \left( \left\lfloor \frac{d_{ij} - J_{ij}}{T_i} \right\rfloor - 1 \right)_0 \quad (33)$$

As before, the zero subscript indicates that negative values are considered zero (in this case, such an event indicates that none of the jobs starting before  $t_s$  can contribute to the busy period). Thus, the contribution of these jobs is  $nC_{ij}^h$  if  $d_{ij}^h \leq D$  and zero otherwise. We can now compute the contribution of the exception handlers of  $\tau_{ij}$  to the response time of a job of  $\tau_{ab}$  in a busy period of duration  $t$  and deadline  $D$ , when the task whose activation time coincides with the start of the busy period is  $\tau_{ik}$ , using the following function:

---

**Algorithm 1:**  $W_{ijk}^h(t, D)$

---

```

1: sum=0;
2: if  $d_{ij}^h \leq D$  then
3:    $n = \left( \left\lfloor \frac{d_{ij} - J_{ij}}{T_i} \right\rfloor - 1 \right)_0$ ;
4:    $sum \leftarrow sum + nC_{ij}^h$ ;
5: if  $d_{ij} + d_{ij}^h \leq D$  then
6:    $x_i \leftarrow \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor$ ;
7:    $sum \leftarrow sum + x_i C_{ij}^h$ ;
8:    $n_{inst} = \left\lfloor \frac{t - \varphi_{ijk}}{T_i} \right\rfloor$ ;
9:   for  $1 \leq p \leq n_{inst}$  do
10:     $De = \varphi_{ijk} + (p-1)T_i + d_{ij} + d_{ij}^h$ ;
11:    if  $De \leq D$  then
12:       $sum \leftarrow sum + C_{ij}^h$ ;
13: return sum;
```

---

Thus, we can modify Equation (15) from Section 5.2 to:

$$\begin{aligned}
W_{ijk}(t, D) = & \left( \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \min \left( \left\lfloor \frac{t - \varphi_{ijk}}{T_i} \right\rfloor, \left\lfloor \frac{D - \varphi_{ijk} - d_{ij}}{T_i} + 1 \right\rfloor \right) \right)_0 (C_{ij} + s) \\
& + W_{ijk}^h(t, D)
\end{aligned} \tag{34}$$

Also, we need to modify the critical instants to be examined in order to accommodate the inclusion of the exception handlers in the busy period, thus, Equation (20) becomes:

$$\Psi = \bigcup \{ \varphi_{ijk} + (p-1)T_i + d_{ij} \} \bigcup \{ \varphi_{ijk} + (p-1)T_i + d_{ij} + d_{ij}^h \} \tag{35}$$

$$\forall p = P_{0,ijk} \cdots P_{L,ijk}, \quad \forall j, k \in \Gamma_i$$

Similarly, Equation (7) needs to be modified to:

$$L = \sum_{\forall i, j} \left\lceil \frac{L + J_{ij}}{T_i} \right\rceil (C_i + m_i s + s + C_{ij}^h) \tag{36}$$

Essentially extending the execution time of a job by  $C_{ij}^h$  because, in the worst-case, the exception handler is triggered at the last instant of time in the execution of the job, thus, placing a demand on the processor equal to the total time of the job and the handler. The rest of the analysis remains unchanged.

## 7. DYNAMIC JITTER AND OFFSETS

In this section we briefly indicate how the iterative techniques first developed by Palencia and Harbour in [18], based on Tindell and Clark's Holistic analysis [23], and later improved in [7, 17, 19] can be used to provide response time analysis of distributed systems programmed using STM. From the analysis in Sections 5.2 and 6.1, we can perform response time analysis of systems where concurrency control is managed using STM, tasks have offsets and jitters, and failures are possible. Initially, the offset of each task is set to the minimum possible completion time of its predecessor, i.e.,

$$\phi_{ij} = \sum_{i \leq k \leq j} (\delta_{ik} + C_{ik}) + \delta_{ij} \quad \forall 1 \leq j \leq N_i \tag{37}$$

where  $\delta_{ij}$  is the communication delay between nodes  $i$  and  $j$ . The jitters are all set to zero and the response time,  $R_{ij}$ , is computed using either the analysis in Section 5.2, if failures are not being considered, or Section 6.1, if we wish to consider failures. Then, jitters are modified as follows:

$$J_{i1} = 0 \tag{38}$$

$$J_{ij} = R_{ij-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \leq N_i \tag{39}$$

Offsets remain unchanged. Essentially, this means that the jitters are modified so that each task,  $\tau_{ij}$ , is released at most  $\delta_{ij}$ , the communication delay, time units after the completion of its predecessor  $\tau_{ij-1}$ . We then compute the response times again using either the analysis in Section 5.2 or 6.1. This process is repeated until the result of two successive iterations are the same, at which point we have obtained the response time for each task. If the response times do not diverge, the process above is guaranteed to converge to the solution since the process is monotonic in its parameters.

Naturally, during the computation only the contribution of the tasks running on the same processor is taken into account when computing the response times.

## 8. EXPERIMENTS

In this section, we experimentally evaluate the performance of the proposed algorithm against a system simulated using RTNS [16]. In the first set of experiments, we measure the average ratio,  $R_{ana}/R_{sim}$ , between the response time of the analysis to the response time of the simulation. Execution times and periods are randomly generated as is the number of STM transactions in each task. Figure 3 shows the result of our first set of experiments.

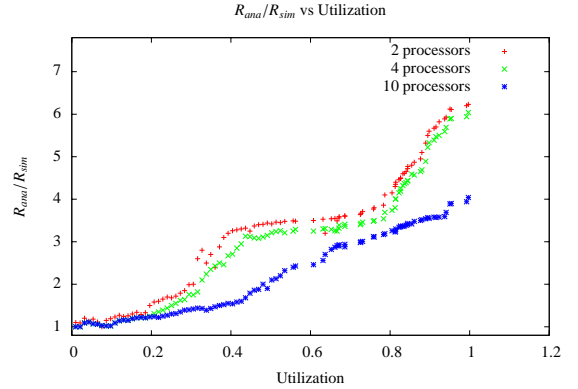


Figure 3: Ratio vs. Utilization

The utilization depicted on the  $x$ -axis is derived from Equation (14). Therefore, ideally, all tasks should meet their deadline for all utilizations at or below one. In this experiment, we studied three different systems. In the first system, only two processors exist and the tasks make remote invocations to either one at random. The utilization, in all three cases we studied, is defined as the *maximum* utilization experienced by any node in the system. The other systems have four and ten processors respectively.

In all our experiments, the response time derived from the proposed analysis is higher than that of the simulation. This can be seen from the fact that the ratio  $R_{ana}/R_{sim}$  never falls below one in Figure 3. Also, the ratio becomes worse as system load increases. This occurs because as the system gets more loaded, the number of interruptions increases and hence the pessimism of the analysis increases (since we assume each interruption will result in a retry). Also, as the number of processors in the system increases, the ratio becomes better. This occurs because the utilization measured on the  $x$ -axis is the *maximum* utilization experienced by any node. Therefore, it is possible for other nodes in the system to be lightly loaded, leading to less interferences and, thus, less pessimism in the analysis.

Also, the pessimism of the analysis depends on the cost of the transactional part of the code,  $s$ , relative to the execution time of the non-transactional part of the code  $C_{ij}$ . The larger the ratio of  $s$  to  $C_{ij}$ , the more pessimistic the analysis becomes, because the pessimism in the proposed analysis is in the number of retries. Increasing the weight of the retries in the analysis by increasing the cost of the transactional component of the code results in larger estimates of worst-case response times. Figure 4 shows the result of an experiment where we compare the performance of a system

to other systems that have a value of  $s$  twice as large and half as large as the base system.

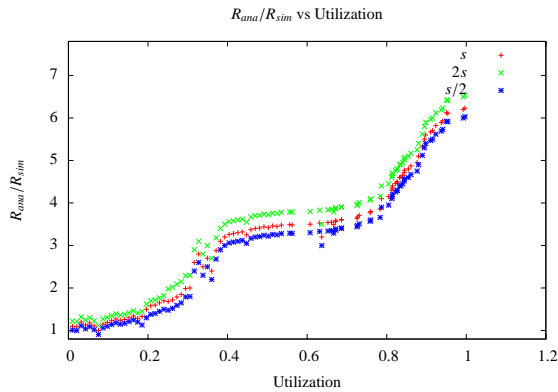


Figure 4: Ratio vs. Utilization

It can be seen that the larger values of  $s$  cause greater divergence from the simulation results.

## 9. CONCLUSION

In this paper we presented an algorithm for computing an upper bound on the response time of tasks in a distributed real-time system where concurrency control is managed using STM and nodes are subject to crash failures. We compared the result of our analysis to a simulation of the system in order to determine the efficacy of the proposed solution.

The result of this study indicates that it is possible to provide timeliness guarantees for distributed systems programmed using STM. This allows for the first time, the usage of STM as a concurrency control mechanism (among others) for programming distributed real-time systems. Future research includes studying the different approaches for incorporating STM as outlined in [6] and dealing with some of its shortcomings. For example, we can consider whether buffered I/O can be used in STM code blocks and whether it is possible to eliminate the problem of weak atomicity using declarative languages or placing restrictions on the use of variables in imperative programming languages. Other areas of research include reducing the overhead of STM using software-hardware hybrid techniques and implementation optimizations. We also plan to consider aperiodic tasks and overload scheduling (i.e., providing assurances during overload conditions).

## 10. REFERENCES

- [1] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *IEEE RTSS*, pages 92–105, Dec 1996.
- [2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *IEEE RTSS*, pages 28–37, Dec 1995.
- [3] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*. Amsterdam: Kluwer Academic Publishers., 1997.
- [4] J. Bobba, R. Rajwar, and M. Hill. Transactional memory bibliography. <http://www.cs.wisc.edu/trans-memory/biblio/swtm.html>.
- [5] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, New York, NY, USA, 2008. ACM.
- [6] S. F. Fahmy, B. Ravindran, and E. D. Jensen. On scalable synchronization for distributed embedded real-time systems. In *SEUS*, October 2008. <http://www.real-time.ece.vt.edu/seus08.pdf>.
- [7] M. G. Harbour and J. C. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *IEEE RTSS*, page 200, 2003.
- [8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [9] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
- [10] P. Holman and J. H. Anderson. Supporting lock-free synchronization in pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66(1):47–67, 2006.
- [11] M. H. Klein, T. Ralya, B. Pollak, R. Obeniza, and M. G. Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [12] T. F. Knight. An architecture for mostly functional languages. In *Proceedings of ACM Lisp and Functional Programming Conference*, pages 500–519, Aug 1986.
- [13] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [14] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. ACM Press, Mar 2006.
- [15] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time java. *RTSS*, 0:62–71, 2005.
- [16] P. Pagano, P. Batra, and G. Lipari. A framework for modeling operating system mechanisms in the simulation of network protocols for real-time distributed systems. *IPDPS*, 0:160, 2007.
- [17] J. Palencia and M. G. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. *ECRTS*, 00:3, 2003.
- [18] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the 19th IEEE RTSS*, pages 26–37, 1998.
- [19] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. *RTAS*, pages 66–75, 2005.
- [20] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.
- [21] M. Spuri. Analysis of deadline scheduled real-time systems. Technical report, In *Rapport de Recherche RR-2772*, INRIA, 1996.
- [22] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [23] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. 50:117–134, 1994.