Transient Fault Tolerance Patterns for Real Time Systems with Arbitrary Deadline

Smriti Agrawal, Rama Shankar Yadav, Ranvijay

Abstract— Occurrence of transient faults have increased as the chip area is reduced in the mobile devices. Tolerance to transient faults is become even more essential for a healthy real time system. This paper presents a two phase preemption control pattern approach to improve average tolerance to transient faults. It also improves the acceptability domain by accepting task sets with arbitrary deadline and some task sets previously rejected by existing techniques. Phase-1 employs checkpointing based tolerance patterns. The phase-2 uses preemption control technique to improve the quality of service. Theorems and lemmas are derived to ensure the correctness of pattern and feasibility of the task set. A new burst tolerance parameter is proposed to rate the performance of offline fault tolerance scheduling strategies. The proposed patterns perform better than existing ones, for both synthesized and real-world applications in terms of average and burst tolerance.

Index Terms— Arbitrary deadline, burst of faults, checkpointing, fault tolerance, preemption overhead, preemption control, real time systems, scheduling.

1 INTRODUCTION
Fault tolerance is defined as the ability of a system to comply with its specification despite the presence of faults (transient or permanent type) in any of its components [2, 3]. Transient faults occur due to environmental effects such as temperature, pressure, humidity etc. Tolerance to these faults is critical for the working of Real-Time Systems. In a real-time system correct computation of a result must finish before its specified deadline [1]. Present work proposes techniques for transient fault tolerance in real-time systems.

2 RELATED WORK
Tolerance to transient faults can be achieved through roll-forward or roll-back techniques [4, 5, 6, 3]. In roll-forward approach, copies of a task are executed in parallel on redundant hardware and results are voted upon. In case of a disagreement in the results, faulty processor (minority) is updated to correct state as computed by majority processors. The roll-back technique re-executes faulty portion requiring only time redundancy to tolerate transient faults. The requirement of redundant hardware in roll-forward leads to increment in cost, weight, size and energy requirement of the system. On the other hand, roll-back recovery would require lesser hardware as it needs only time redundancy to tolerate transient faults. Thus, roll-back technique is most suited for portable devices. During execution of a task if a fault occurs it must restart (roll-back) to its initial state and redo the whole execution. This roll-back time can be reduced by use of checkpointing scheme [4, 5, 6] where intermediate results are verified at checkpoint. In case of fault recalculation must be done up to last checkpoint only. Refer figure 1, where roll-back portion is up to last checkpoint (i.e., third checkpoint) only. However, checkpointing itself incurs an overhead, several techniques such as checkpoint buffering with copy-on-write [7, 8], compression [9], mirror copy [10], DRAM [11] have been implemented to reduce it [12, 13]. Authors [14, 15], suggested Lemma, to show the condition where checkpointing scheme could be beneficial over non-checkpointing. The interval between two checkpoints can be equidistant [11] or adaptive [16, 17, 18]. However, equidistant checkpointing is optimal for offline scheduling [19] and this paper uses the same. Many models in the literature deal with fault tolerance using checkpointing for all type of systems [10, 20, 21, 1, 10, 22, 23, 24]. But all these approaches simplify the problem of real time scheduling by assuming that deadline of each release is less than or equal to its period (referred to as conservative deadline). This assumption prevents a release to wait because of its own kind. Present work makes no such assumption and realistically allow the deadline and period of a task to be independent of each other (arbitrary deadline).

Authors [25] provide a detailed description of a robotics application with arbitrary deadline periodic tasks. Authors [26, 27, 28, 27, 29, 28, 30, 31] have formulated feasible scheduling policies for arbitrary deadline task set for both uniprocessor and multiprocessor systems, but they have not considered possible occurrences of faults. This paper aims to provide an offline fault tolerant scheduling algorithm for periodic task sets with arbitrary deadline on a uniprocessor system. This technique can also applied to multiprocessor systems after load balancing as it is essentially an offline technique. The observations of a system indicate that faults arrival varies between \([0, k_{obs}]\) where \(k_{obs}\) is peak observed faults. Best QoS is achieved, if tolerance of all releases of a task is \(k_{obs}\) and degraded but acceptable quality if \(k_{min} \leq k_{obs}\). Present work aims to achieve best QoS while maintaining acceptable QoS by ensuring tolerance is never less than \(k_{min}\).

Theorem 1 along with lemma 3 ensures that minimum acceptable quality is received, in all situations for single task system. They are further extended for multiple tasks set as theorem 2 and lemma 4. A two phase preemption control

---

Authors

- Smriti Agrawal. E-mail: agrawal.smriti@gmail.com
- Rama Shankar Yadav is currently professor in Department of Computer Science and Engg., MNNIT, Allahabad, India, rsy@mnnit.ac.in
- Ranvijay is currently Assistant Professor in Department of Computer Science and Engg., MNNIT, Allahabad, India, ranvijay@mnnit.ac.in
pattern approach is proposed. In first phase formulation of fault tolerance patterns based on availability of slack time where tolerance of a release can vary from \( k_{\min} \) to \( k_{\text{obs}} \) is achieved using checkpointing. Second phase, further improves average tolerance of the system by preemption control scheme which judiciously avoid preemption of the lower priority release wherever possible to reduce losses incurred as preemption overheads. Thus, besides providing better average tolerance, this work improves application domain by accepting task sets previously rejected by existing techniques [10, 20, 21, 1, 32, 22, 23, 33].

Another major contribution of this work is a QoS parameter called Burst Tolerance. Burst Tolerance evaluates the fault tolerance of offline fault tolerance approaches. This prediction is done offline to estimate the number of faults outburst a technique can handle gracefully. Performance of proposed algorithms is evaluated for synthesized as well as for benchmarked real world applications such as computerized numerical control (CNC) [34], inertial navigation system (INS) [35, 36], generic aviation platform (GAP) [37] and webphone (WP) [38]. The simulations indicate that proposed two phase preemption control pattern approach increases quality of service of the system in terms of providing better average tolerance, burst tolerance.

3 System Modeling

This system model deals with fixed priority preemptive deadline monotonic scheduling of independent periodic task set \( T = \{ \tau_1, \tau_2, \tau_3 \ldots \tau_n \} \) with arbitrary deadlines. Each task \( \tau_i \) has the attributes, worst-case execution time (\( \epsilon_i \)), period (\( p_i \)), relative deadline (\( d_i \)) and a minimum tolerance requirement of \( k_{\min_i} \).

A task may suffer multiple transient faults varying in the range of \([0, k_{\text{obs}}]\) where \( k_{\text{obs}} \) are the utmost observed faults. Without loss of generality, it is assume, that if all releases of a task \( \tau_i \) are able to tolerate at least \( k_{\min_i} \) (\( 0 \leq k_{\min_i} \leq k_{\text{obs}} \)) faults then an acceptable quality of service is achieved. For providing fault tolerance, equidistant checkpointing is used. Checkpointing and preemption overheads are assumed to be constant and independent of task [39, 40, 11].

A task may have various factors affecting its execution time such as branching, looping, cache miss etc. Thus, the worst-case execution time is the utmost time a task will take to complete its execution under fault free scenario. This value can be estimated by code profiling and statistical prediction [41, 42, 43]. Henceforth, execution time of a task refers to the worst-case execution time.

3.1 Terms used

- **Checkpoint saving cost** \((C_s)\): time required to test and save current status to a reliable storage.

- **Checkpoint retrieval cost** \((C_r)\): time required to reload the system with values last tested to be correct from a reliable storage.

- **Checkpoint Overhead** \((C)\): sum of checkpoint saving and retrieval cost, i.e., \( C = C_s + C_r \).

- **Preemption Overhead** \((R)\): context switching time required when a higher priority preempts a lower priority task.

- **Optimal number of checkpoints** \((m_i)\): The number of checkpoints that reduces a task’s total time requirement to its least value, i.e., if number of checkpoints increased further, the total time requirement of the task will not be reduced.

- **Checkpoint interval or recovery block** \((r_b)\): for task \( \tau_i \) it is the sum of the time interval between two consecutive checkpoints and a checkpoint overhead, given as \( e_i/(m_i + 1) + C \), where \( m_i \) is number of checkpoints used by task \( \tau_i \).

- **Release Time** \((R_{el})\): is the time at which a release \( \tau_i^j \) is ready for execution, i.e., \( R_{el}^j = j \times p_i \).

- **Absolute deadline** \((D^j)\): is the time by which a release \( \tau_i^j \) must finish its execution, i.e., \( D^j = j \times p_i + d_i \).

- **Total time requirement** \((R_i(k_i,m_i))\): of a task \( \tau_i \) with \( k_i \) fault tolerance is mathematically, \( R_i(k_i,m_i) = e_i + (m_iC_s) + k_i r_b \).

- **Checkpoint saving cost** \((C_s)\): is the time required to save current status to a reliable storage.

- **Checkpoint retrieval cost** \((C_r)\): is the time required to reload the system with values last tested to be correct from a reliable storage.

- **Checkpoint Overhead** \((C)\): is the sum of checkpoint saving and retrieval cost, i.e., \( C = C_s + C_r \).

- **Preemption Overhead** \((R)\): is the context switching time required when a higher priority preempts a lower priority task.

**Burst Tolerance**

- **Release Time** \((R_{el})\): is the time at which a release \( \tau_i^j \) is ready for execution, i.e., \( R_{el}^j = j \times p_i \).

- **Absolute deadline** \((D^j)\): is the time by which a release \( \tau_i^j \) must finish its execution, i.e., \( D^j = j \times p_i + d_i \).

- **Total time requirement** \((R_i(k_i,m_i))\): of a task \( \tau_i \) with \( k_i \) fault tolerance is mathematically, \( R_i(k_i,m_i) = e_i + (m_iC_s) + k_i r_b \).

**Optimal number of checkpoints** \((m_i)\): The number of checkpoints that reduces a task’s total time requirement to its least value, i.e., if number of checkpoints increased further, the total time requirement of the task will not be reduced.

**Checkpoint interval or recovery block** \((r_b)\): for task \( \tau_i \) it is the sum of the time interval between two consecutive checkpoints and a checkpoint overhead, given as \( e_i/(m_i + 1) + C \), where \( m_i \) is number of checkpoints used by task \( \tau_i \).

**Release Time** \((R_{el})\): is the time at which a release \( \tau_i^j \) is ready for execution, i.e., \( R_{el}^j = j \times p_i \).

**Absolute deadline** \((D^j)\): is the time by which a release \( \tau_i^j \) must finish its execution, i.e., \( D^j = j \times p_i + d_i \).

**Total time requirement** \((R_i(k_i,m_i))\): of a task \( \tau_i \) with \( k_i \) fault tolerance is mathematically, \( R_i(k_i,m_i) = e_i + (m_iC_s) + k_i r_b \).

Mathematically, \( ft_i = max(0, ft_i^{j-1} - rel_i^j) \).

**Response Time** \((R_e^j(k_i,m_i))\): is the difference between finish time and release time of a release \( \tau_i^j \). Mathematically, \( R_e^j(k_i,m_i) = ft_i^j - rel_i^j \).

**Fault tolerance pattern**: is a sequence of a tolerance guaranteed to a set of releases of a task which repeats itself.
3.2 Checkpointing Scheme [14, 15]

Consider a periodic task \( \tau_i \) tolerating \( k_i \) faults where \( k_{i3} \) faults may occur during checkpoint saving, \( k_{i2} \) during checkpoint retrieval and \( k_{i3} \) during non-checkpoint execution [11]. The amount of rollback required will be \( k_{i3}(e_i + C_s) \), \( k_{i2}C_r \) and \( k_{i3}(e_i + C_r) \) as seen in the figure 2. It is maximum for the case faults occur during checkpoint saving. Thus, requirement time to tolerate \( k_i \) faults is worst when all faults occur during checkpoint saving (i.e., \( k_1 = k_{i1} \), \( k_2 = k_{i2} = 0 \)) will be \( R_i(k_i, m_i) = e_i + (m_iC_s) + k_i(C_s + C_r) + (k_ie_i/(m_i + 1)) \)…… (1)

For a given task \( \tau_i, e_i, C_s, C_r \) are constant and \( m_i \) is responsible for variation of requirement time for tolerating \( k_i \) faults. The lemma 1 and lemma 2 state the condition for optimal number of checkpoints for a task, in order to tolerate \( k_i \) faults.

**Lemma 1** [14, 15]. The optimal number of checkpoints for a task \( \tau_i \) with execution time \( e_i \), guaranteeing \( k_i \) level fault tolerance, checkpoint storing \( C_s \) and retrieval cost \( C_r \) is \( m_i = \max\{0, \sqrt{k_i e_i/C_r} - 1\} \).

**Lemma 2** [14, 15]. For a task \( \tau_i \) with execution time \( e_i \), guaranteeing \( k_i \) level fault tolerance, checkpoint storing \( C_s \) and retrieval time \( C_r \), checkpointing scheme is beneficial over non-checkpointing if \( \max(C_s, C_r) \leq k_i e_i/(\sqrt{2k_i e_i} + 1)^2 \).

4 PATTERNS FOR TASKS WITH ARBITRARY DEADLINES.

Newer technologies operate at lower noise margins, leading to higher transient fault arrival rate [11, 14, 33]. This work is an extension of [11, 33] in which authors presume that acceptable quality of service would be achieved if all releases of a task guarantee tolerance to \( k_{\min} \) faults. They further assume that task sets have conservative deadline. We refer to their tolerance pattern as Kmin_Pattern.

The proposed work aims to achieve best quality of service by ensuring tolerance to utmost number of observed \( k_{obs} \) faults to all releases. However, if \( k_{obs} \) is high then a fault tolerant scheme that provides deterministic guarantee may not exist. In such a situation the acceptable quality of service has to be honored, failing which the task set is rejected. We also relax the conservative deadline assumption of [11] and address the task sets with arbitrary deadlines with measurable preemption overhead.

A two phase preemption control pattern approach is proposed. In the first phase we formulate fault tolerance patterns based on the availability of the slack time, where the tolerance of a release can vary from \( k_{\min} \) to \( k_{obs} \). While, the second phase, further improves the average tolerance of the system by preemption control scheme to reduces the losses due to preemptions. Moreover, by controlling preemption, the task sets failing to achieve acceptable quality with existing approaches [11] may be accepted.

4.1 Conceptual Framework

The total requirement time \( R_i(k_i, m_i) \) (refer equation (1)) is directly proportional to the number of faults it can tolerate. For simplicity of explanation suppose there is only one task in the system whose deadline is greater than its period and it guarantee to tolerate \( k_i \) number of faults such that \( R_i(k_i, m_i) > (p_i) \). The next release which arrives at \( (p_i) \) is forced to wait for \( \delta = R_i(k_i, m_i) - p_i \). Refer figure 3. The subsequent releases wait for \( 2\delta, 3\delta \ldots n\delta \) and eventually, a release with \( n\delta + R_i(k_i, m_i) > (d_i) \) will fail.

Theorem 1 along with the lemma 3 assure that the minimum quality of service will be achieved.

**Lemma 3** [14, 15]. For a single task system \( \tau_i \) with execution time \( e_i \), period \( p_i \) and relative deadline \( d_i \) can always tolerate \( k_i \) faults, if and only if its computation requirement with \( k_i \) fault tolerance is less than or equal to the minimum of its relative deadline and period i.e., \( R_i(k_i, m_i) \leq \min(d_i, p_i) \).

**Proof.** For a task \( \tau_i \) to be feasible the utilization of the task must always be less than or equal to one i.e., \( u_i = R_i(k_i, m_i)/\min(d_i, p_i) \leq 1 \) [44] therefore, for all releases to tackle \( k_i \) faults the time \( R_i(k_i, m_i) \) should always be less than or equal to \( \min(d_i, p_i) \) i.e., \( R_i(k_i, m_i) \leq \min(d_i, p_i) \).

**Theorem 1.** A uniprocessor system comprising of a single task with attributes \( e_i, p_i, d_i, k_{\min} \) if its \( j \)th release finishes by its absolute deadline \( D_j^i \) then the next release will be schedulable
with at least \( k_{\text{min}} \) tolerance.

**Proof.** Proof is given as an Appendix 1.

Theorem 1 ensures that a release may be allowed to finish by its deadline while preserving the minimum tolerance guarantee to the next release. Thus, a release may improve its tolerance and go up to its deadline. The best tolerance a release can provide is referred to as \( k_{\text{max}}(\leq k_{\text{abs}}) \) estimated using function calculate_\( k_{\text{max}} \) stated in Appendix 2.

Computation requirement with \( k_{\text{max}} \) tolerance will be \( R_i(k_{\text{max}}, m_{\text{max}}) \leq d_i \). Then best quality of service equivalent to \( k_{\text{max}} \) tolerance can be provided, if the deadline of this task is less than or equal to its period \( d_i \leq p_i \). This is because, as suggested by lemma 3, utilization \( u_i \) would be \( R_i(k_{\text{max}}, m_{\text{max}})/p_i \) which is always less than or equal to one. However, if both deadline and required time are greater than period \( (p_i < R_i(k_{\text{max}}, m_{\text{max}}) \leq d_i) \) then utilization \( u_i \) would be \( R_i(k_{\text{max}}, m_{\text{max}})/p_i \) which is always greater than one, thus all releases cannot be granted \( k_{\text{max}} \)-tolerance, leading to a variable tolerance in the range of \( [k_{\text{min}}, k_{\text{max}}] \).

This variation in tolerance is referred to as patterns described in the following section and are formulated in the first phase. In second phase preemption control is applied. The preemption control technique is neither purely preemptive nor non-preemptive rather it is judicious preemption reducing the losses due to preemption.

Further, the Burst Tolerance parameter is estimated to predict the outburst number of faults a technique gracefully handles even though it is not designed for it.

**K\text{min-Kmax Pattern for Single Task}**

To improve the QoS, the tolerance of a release can be increased but it in turn increase its requirement time. If the requirement time at a tolerance level \( k_{\text{max}} \) is beyond its period, (possible only when \( d_i > p_i \)) by amount \( \delta = R_i(k_i, m_i) - p_i \), (where \( k_i = k_{\text{max}} \)). Then in general, \( x^{th} \) release has to wait for \( (x - 1) \times \delta \) (if all \( x - 1 \) releases go beyond the period). The finish time will be \( f(t_i(k_i, m_i)) = R(t_i) + R_i(k_i, m_i) + (x - 1) \times \delta \) which maybe more than the absolute deadline of the task consequently this release would be infeasible. But by theorem 1, if \( (x - 1)^{th} \) release was feasible then \( x^{th} \) release will be feasible with atleast minimum tolerance. Here, we first discuss the case where a release can either tolerate acceptable \( k_{\text{min}} \), number of faults or maximum \( k_i = k_{\text{max}} \) (known as K\text{min-Kmax Pattern}). If \( (x - 1)^{th} \) release finishes by its deadline then time available for execution for \( x^{th} \) release before its deadline will be \( p_i \) (by equation (11)). If it is scheduled with minimum tolerance \( R_i(k_{\text{min}}, m_{\text{min}}) \) (where \( (R_i(k_{\text{min}}, m_{\text{min}}) + \Delta = p_i) \) by lemma 3) then it will finish before its deadline by an amount \( \Delta \). Hence, the time available for the next release will be \( p_i + \Delta \). In other words, the wait time of \( (x + 1)^{th} \) release will be reduced by \( \Delta \). If \( (x + 1)^{th} \) release is made minimum tolerant then wait time of \( (x + 2)^{nd} \) release will reduce by \( 2 \times \Delta \). Thus, for a task every maximum tolerant release overload the system by an amount \( \delta \) while a minimum tolerant release will lighten it by \( \Delta \). This can be understood by the example 1

**Example 1:** Consider a task \( \tau_i \) with attributes \( (e_i, p_i, d_i, C_i, C_r, k_{\text{min}}) < 10, 20, 40, 0.5, 0.5, 1, 1 > \).

To tolerate \( k_{\text{min}} \) faults, requirement time \( R_i(k_{\text{min}}, m_{\text{min}}) = 15 \) refer equation (1). Since, \( R_i(k_{\text{min}}, m_{\text{min}}) \leq \min (d_i, p_i) \) then all the releases of the task is feasible with acceptable quality of service, refer lemma 3. The best tolerance that can be granted is \( k_{\text{max}} = 13 \) where \( R_i(k_{\text{max}}, m_{\text{max}}) = 38.64 \).

Whenever a release guarantees tolerance of \( k_{\text{max}} = 13 \) then it will overload the system by an amount \( \delta = R_i(k_{\text{max}}, m_{\text{max}}) - p_i = 38.64 - 20 = 18.64 \). On the other hand, tolerance of \( k_{\text{min}} = 1 \) will relieve the system by an amount \( \Delta = p_i - R_i(k_i, m_i) = 20 - 15 = 5 \).

Now, the first release \( R_0 \), with release time \( R(t_0) = 0 \) and absolute deadline of \( D_0 = 40 \), has zero waiting time. If tolerance of \( k_{\text{max}} = 13 \) is granted to it the finish time is \( f(t_0(k_{\text{max}}, m_{\text{max}})) = 0 + R_i(k_{\text{max}}, m_{\text{max}}) + 0 = 38.64 < 40 \).

Since, this release is feasible this tolerance is granted to it. But the second release \( t_1 \) released at \( R(t_1) = 20 \) and has an absolute deadline of \( D_1 = 60 \) has waited from time \( 20 = 38.64 \).

Thus, if it is granted a tolerance of \( k_{\text{max}} = 13 \) then its finish time will be \( f(t_1(k_{\text{max}}, m_{\text{max}})) = R(t_1) + R_i(k_{\text{max}}, m_{\text{max}}) + \delta = 77.28 > 60 \), hence, this release would fail. Thus, decreasing the tolerance of this release to acceptable i.e., \( k_{\text{min}} = 1 \), then finish time will be \( f(t_1(k_{\text{min}}, m_{\text{min}})) = 53.64 < 60 \). The subsequent release \( t_2 \), will be released at \( R(t_2) = 40 \), and will wait for \( 53.64 - 40 = 13.64 \).

Note, that the first release has overloaded the system by an amount \( \delta = 18.64 \), forcing the second release to wait for 18.64. The wait time of third release is \( \Delta = 5 \) units less than its previous release, i.e., 13.64. Hence, the each minimum tolerant release would relieve the by \( \Delta = 5 \), and finally at time \( 98.64 \) the system would be idle till time \( 100 \). The sixth release \( t_5 \), released at \( R(t_5) = 100 \) and has absolute deadline of \( D_6 = 140 \), can again tolerate \( k_{\text{max}} = 13 \), faults. Thus, a fault tolerance pattern of 13, 1, 1, 1, 1 is received. For more details refer table 1.

It can be observed from the example 1 shown in table 1 that in case, the second release fails to guarantee 13 faults as its deadline would expire, the K\text{min-Kmax Pattern} forces it to tolerate minimum faults. However, the theorem 1 says that this release can have a tolerance at least \( k_{\text{min}} \). Thus, there is a possibility that this release could provide an enhanced tolerance guarantee leading to a greedy pattern called K\text{Pattern} discussed in the next subsection.

**K\text{Pattern for Single Task}**

For K\text{Pattern} if a release is incapable of tolerating \( k_{\text{max}} \) faults then instead of directly forcing it to tolerate \( k_{\text{min}} \) faults, an intermediate level of tolerance \( (k_i \text{ between } k_{\text{min}} \text{ and } k_{\text{max}}) \) is chosen. For example the task in the table 1; the second release \( (D_1 = 60) \) cannot tolerate 13 faults then it is granted an intermediate tolerance of 3 (finish time 58.89 < 60) instead to...
forcing it to tolerate 1 ($k_{\text{min}}$) fault as was done in Kmin_Kmax_Pattern.

A general function to estimate the tolerance patterns (calculate_fault_tol_pattern) is stated in Appendix 2.

The patterns in the table 1 can be obtained from the above function by considering a task $t_i$. Kmin_Kmax_Pattern is able to provide $k_{\text{max}}$ guarantee to some of the releases because it has fast recovery mechanism by switching to $k_{\text{min}}$ tolerance. The K_Pattern provides a uniform approach by settling around $k_1$ and $k_2$ tolerance where $R_i(k_2, m_2) \leq \min(d_i, p_i)$ and $p_i < R_i(k_2, m_2) \leq d_i$. On the other hand, the existing Kmin_Pattern [11] would provide only the acceptable tolerance of $k_{\text{min}}$. However, to give it a fair chance for comparison its tolerance is improved to the best possible which would be $k$ (such that $R_i(k_1, m_1) \leq \min(d_i, p_i)$) this shown in table1.

### Burst Tolerance for Single Task

This paper proposes an offline scheduling that improves the QoS in terms of fault tolerance. However, the exact time of occurrence of failure can never be predicted. The actual number of faults occurring online may be less than, equal to or more than the tolerance guaranteed. When scheduling is done offline it is incapable of updating itself as per the systems online requirement. Thus, if the number of faults that actually occur are less than or equal to the tolerance guaranteed the system survives. However, in case a release encounters an outburst of faults (more than expected) then possibility of its surviving while maintaining guaranteed tolerances to the remaining releases is measured as burst tolerance of the system.

Several offline techniques [14, 15, 38, 45, 46, 47, 33] exists that provide the required tolerance guarantee. However, the parameter burst tolerance can be used to differentiate among these as it measures the survivability chances in case some releases encounter more than the expected faults.

The prediction is done offline with the assumption that if all other releases tolerate guaranteed number of faults ($k_{\text{min}}$) then the burst number of faults this release will be able to survive through. Formally,

**Definition 1. The maximum number of faults a release can tolerate (with its rollback time governed by the number of checkpoints applied to it in the offline scheduling) such that it neither misses its own deadline nor forces any lower priority release to do so with minimum guaranteed tolerance ($k_{\text{min}}$) is called the burst tolerance of a release.**

Consider the $j^{th}$ release of a task $t_i$ with attributes $e_i$, $p_i$, $d_i$, $k_{\text{min}}$, $k_i$, $m_i$ and $b_i$ where $k_i$ is the tolerance granted to the $j^{th}$ release by the pattern, $m_i$ is the number of checkpoints deployed in the release offline and $b_i$ is the actual number of faults which occur during online. Thus, for each fault occurred the total time for recovery will be $rb_i(m_i) = (e_i/(m_i + 1)) + (C_s + C_c)$. If $b_i$ faults actually occur in the $j^{th}$ release then the requirement of this release will be $B_i(m_i, b_i) = \xi_i(m_i) + b_i \cdot rb_i(m_i)$ where $\xi_i(m_i) = e_i + (m_i * C_s)$, instead of $R_i(k_i, m_i)$ (equation (1)). Thus, a gain of $G_i = R_i(k_i, m_i) - B_i(m_i, b_i)$, which can be positive, negative or zero depending on the values of $b_i$ is less than, greater than or equal to $k_i$. If $b_i = k_i$, i.e., actual number of fault that occur are same as that tolerance granted then gain $G_i$ is zero and the pattern remains same. However, if lesser number of faults occurred than tolerance guaranteed then $G_i$ (is positive) amount of time is saved. A negative value of $G_i$ indicates that this release will overload the system by the same amount.

Finish time after $b_i$ faults occur is $ft_{j+1} = Rel_{j+1} + B_i(m_i, b_i) + \max(0, ft_{j+1} - Rel_{j+1})$, i.e., $= Rel_{j+1} + R_i(k_i, m_i) - G_i + \max(0, ft_{j+1} - Rel_{j+1})$. The next, $(j + 1)^{th}$ release will wait for $W_{j+1} = \max(0, ft_{j+1} - Rel_{j+1})$. The total time available to any release to execute is $d_i$ out of which it waits $W_{j+1}$ and has a minimum requirement of $\xi_i(m_i)$ thus almost number or rollbacks it can take or in other words, the burst of faults the $(j + 1)^{th}$ release would tolerate without rendering the system to be infeasible will be $b_{j+1} = [(d_i - W_{j+1} - \xi_i(m_i))]/rb_{j+1}(m_i)$.
Continuing with the example 2 discussed in the table 1, the burst measurement can be observed in table 2. For Kmin_Kmax_Pattern the first release \( r^1 \) is 13 fault tolerant, if 13 faults actually appear then no gain. However, if lesser than 13 number of faults actually appear during its online execution then it will finish earlier. For prediction in offline it is assumes that it will never encounter lesser than \( k_{min_i} = 1 \) fault. In such case, \( B^i(m_i, b_i) = \xi_i(m_i) + b_i \cdot r^i(b_i) \cdot m_i = 18 + 1 \cdot 1.588 = 19.588 \). The second release \( r^2 \) would be released at time \( Rel^2 = 20 \) and has an absolute deadline of \( D^2 \) = 60 and guarantees tolerance of \( k_{min_i} = 1 \) faults with \( m_i = 4 \) number of checkpoints. Since, the first release finishes at time=19.588, the system would remain idle till time=20. Hence, wait time for second release is \( W^i = 0 \), so its burst tolerance will be \( bt^i = [(d_i - W^i - \xi_i(m_i))/r^i(b_i(m_i))] = [(40 - 0 - 12)/3] = 9 \). This indicates that the second release will be able to tolerate a burst of 9 faults compared to guaranteed 1 fault (during Kmin_Kmax_Pattern formation).

Similarly, if actual number of faults that occur during both first and second release is \( k_{min_i} = 1 \), then they finish at time 19.588 and 35 respectively. Hence, the third release having \( m_i = 4 \) number of checkpoints will survive through a burst of \( bt^i = [(40 - 0 - 12)/3] = 9 \). Thus, burst tolerance for Kmin_Kmax_Pattern is min(13,9) = 9 and 11 (min(13,11)) for K_Pattern (refer table 2). However, the performance of the K_Pattern is worst with burst tolerance of 2.

### 4.2 Multiple Task Model

This section extends the single task approach to set of independent periodic tasks \( T = \{\tau_1, \tau_2, \tau_3 \ldots \tau_n\} \) with attributes same as those considered for a single task. The priorities are assigned based on deadline monotonic and indices are assigned accordingly. In multiple task system, a release may be preempted by higher priority releases incurring an overhead to save and retrieve its context to and from a reliable storage. This overhead is called preemption overhead (8). However, always disallowing a release from preempting a lower priority release may result into missed deadlines. In this work instead of always allowing/disallowing preemption, we suggest a judicious preemption scheme. That is, whenever a higher priority release can wait without missing its deadline the preemption of the lower priority release is avoided. However, if deadline of higher priority release forces preemption then the preemption is allowed with overhead.

This work proposes a two phase preemption control pattern approach. In the first phase patterns are formulated as was done for single task. While, the second phase, performs preemption control to further improve the tolerance by utilizing the overhead saved by preemption control and idle slots.

**Phase-1: Pattern Formation for Multiple Task Set**

For a multiple task model, finish time of release \( \tau^i \) [11] (refer section II terms used) is the sum of its release \( Rel^i \) and response time \( (Re^i(k_i, m_i)) \). The response time \( (Re^i(k_i, m_i)) \) of \( \tau^i \) will at least be equal to its requirement, i.e., \( R^i(k_i, m_i) \).

However, it will be forced to wait, while a higher priority release of tasks \( \tau_j \) to \( \tau_{j-1} \) released during \( ([f^i_t/p_n] - [f^i_{t-1}/p_n]) \) executes for their required \( R^j(k_j, m_j) \) time. Each preemption would also incur overhead of \( \delta \). Thus, the total time due to preemption by higher priority tasks would be \((x^i(k_i, m_i) = \sum_{b=1}^{b} ([f^i_t/p_n] - [f^i_{t-1}/p_n]) (R^j(k_j, m_j)) + \delta) \) where \( b \) is the number of higher priority releases preempting \( \tau^i \). When deadline of a task is greater than its period then it may happen that the finish time of the previous release \( f^i_{t-1} \), \( f^i_t \), \( m^i \), may be after the release of \( \tau^i \), i.e., release \( \tau^i \) is forced to wait \( f^i_{t-1} - Rel^i \) on its own account. Thus, the wait time due to \( \tau^i \) will be \((max(0, f^i_{t-1} - Rel^i)) \).

Therefore, response time \((Re^i(k_i, m_i)) \) is;

\( Re^i(k_i, m_i) = R^i(k_i, m_i) + x^i(k_i, m_i) + (max(0, f^i_{t-1} - Rel^i)) \).

The finish time 
\( f^i_t = Rel^i + run^i(k_i, m_i) + \left( max(0, f^i_{t-1} - Rel^i) \right) \). ....(12)

Where \( run^i(k_i, m_i) = R^i(k_i, m_i) + x^i(k_i, m_i) \). Lemma 4 states the condition for tolerating at least \( k_{min_i} \) number of faults by each task in the set.

**Lemma 4** [15]. **Every task \( \tau^i \) belonging to a periodic task set \( T \) can tolerate \( k_i \) faults, if and only if the run time \( run^i(k_i, m_i) \) of \( f^i_t \) release** of task \( \tau^i \) released with all the higher priority tasks (critical instance) is less than or equal to the minimum of the relative deadline and period i.e., \( run^i(k_i, m_i) \leq \text{min}(d, p_i) \).

The proof remains same as that for Lemma 3 with requirement time \((R^i(k_i, m_i)) \) replaced by \( run^i(k_{min_i}, m_{min_i}) \).

The lemma 4 guarantees the minimum desired tolerance to all the releases but as observed for a single task some releases may provide an improved guarantee \( \tau^i \). However, improvement in the tolerance of a release \( \tau^i \) would increase its requirement time \((R^i(k_{min_i}, m_{min_i})) \) to \( R^i(k_i, m_i) \) and hence, its finish time. Moreover, the improvement in the tolerance of a releaser \( \tau^i \) will also increase the finish time of the lower priority releases \( i.e., f^i_t \leq D^i \) and \( f^i < D^i \lor \tau_e \in \{\tau_{i+1}, \tau_{i+2}, \ldots \tau_n\} \) ..............(13)

The maximum value of \( k_i \) (i.e., \( k_{max_i} \)), the best tolerance that can be

### Table 2:

<table>
<thead>
<tr>
<th>Task Attributes</th>
<th>Kmin_Kmax_Pattern</th>
<th>K_Pattern</th>
<th>Kmin_Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j ) (</td>
<td>R^i_t</td>
<td>D^i</td>
<td>k^i</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>40</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>80</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>80</td>
<td>11</td>
</tr>
</tbody>
</table>
can be provided to any release of task $\tau_i$, can be estimated by modifying the function $\text{calculate}_{\text{kmax}}$ stated for a single task (by accommodating the condition stated in equation (13)). As observed for single task model guaranteeing maximum tolerance $k_{\text{max}}$ may not be possible for all the releases. In such situations the tolerance of the failing release and/or the higher priority releases (preempting it) has to be reduced. However, this reduction can be up to the minimum tolerance $k_{\text{min}}$ for acceptable quality of service. This can be guaranteed by theorem 2.

**Theorem 2.** For a given periodic task set $T = \{\tau_1, \tau_2, \tau_3 \ldots \tau_n\}$ if the $j^{\text{th}}$ release of task $\tau_j$ becomes infeasible when higher priority tasks are guaranteed for number of faults (may be more than minimum) this release will be feasible if all higher priority task preempting it and itself are reduced to minimum tolerance.

**Proof.** Proof is given in Appendix 1.

### Table 3:

**FOR TASK SET $T$ (EXAMPLE 2)**

<table>
<thead>
<tr>
<th>Task $\tau_i$ attributes</th>
<th>Kmin_Kmax_Pattern</th>
<th>K_Pattern</th>
<th>Kmin_Pattern</th>
<th>RM_Kmin_Kmax_Pattern</th>
<th>RM_K_Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task $\tau_1$ attributes</td>
<td>$j$</td>
<td>$R_{j}$</td>
<td>$D_{j}$</td>
<td>$k_{j}$</td>
<td>$f_{j}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>60</td>
<td>14</td>
<td>20.18</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>100</td>
<td>14</td>
<td>60.18</td>
<td>72</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>140</td>
<td>14</td>
<td>100.18</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>120</td>
<td>180</td>
<td>2</td>
<td>133.13</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>160</td>
<td>220</td>
<td>2</td>
<td>173.13</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>260</td>
<td>2</td>
<td>213.13</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>240</td>
<td>300</td>
<td>System idle till 240</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And so on...

| Task $\tau_2$ attributes | $j$ | $R_{j}$ | $D_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ | $k_{j}$ | $f_{j}$ | $b_{j}$ |
|---------------------------|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 90 | 13 | 39.89 | 70 | 13 | 39.89 | 71 | 2 | 39.39 | 2 | 6 | 80 | 16 | 6 | 80 | 16 | 6 | 80 | 16 |
| 1 | 60 | 150 | 13 | 79.89 | 83 | 13 | 79.89 | 82 | System idle till 40 | 2 | 119.39 | 34 | 2 | 135.66 | 35 | System idle till 120 | 2 | 148.79 | 12 | System idle till 150 |
| 2 | 120 | 210 | 2 | 146.26 | 45 | 2 | 146.26 | 67 | System idle till 240 |
| 3 | 180 | 270 | 2 | 193.13 | 50 | 2 | 193.13 | 53 | System idle till 240 |
| 4 | 240 | 330 | System idle till 240 |

And so on...

<table>
<thead>
<tr>
<th>Task $\tau_3$ attributes</th>
<th>$j$</th>
<th>$R_{j}$</th>
<th>$D_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
<th>$k_{j}$</th>
<th>$f_{j}$</th>
<th>$b_{j}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>120</td>
<td>13</td>
<td>119.67</td>
<td>71</td>
<td>13</td>
<td>119.67</td>
<td>71</td>
<td>2</td>
<td>26.26</td>
<td>2</td>
<td>6</td>
<td>32</td>
<td>16</td>
<td>6</td>
<td>32</td>
<td>16</td>
<td>6</td>
<td>32</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>170</td>
<td>2</td>
<td>159.06</td>
<td>36</td>
<td>2</td>
<td>159.06</td>
<td>35</td>
<td>System idle till 40</td>
<td>6</td>
<td>72</td>
<td>16</td>
<td>6</td>
<td>72</td>
<td>16</td>
<td>System idle till 120</td>
<td>System idle till 150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>220</td>
<td>2</td>
<td>198.45</td>
<td>48</td>
<td>2</td>
<td>218.5</td>
<td>47</td>
<td>System idle till 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>150</td>
<td>270</td>
<td>2</td>
<td>224.71</td>
<td>46</td>
<td>2</td>
<td>231.63</td>
<td>46</td>
<td>System idle till 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>320</td>
<td>2</td>
<td>237.84</td>
<td>42</td>
<td>2</td>
<td>278.07</td>
<td>45</td>
<td>System idle till 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And so on...
Theorem 2, implies fault tolerance patterns (Kmin_Kmax_Pattern and K_Pattern) for multiple task set can be deduced as was done for single task. The scheduling of the multiple tasks set can be understood by example 2.

**Example 2:** Considering a task set $T = \{ \tau_1, \tau_2, \tau_3 \} = \{(e_1, p_1, d_1, k_{min}^1): (10,40,60,2), (10,60,90,2), (10,50,120,2) \} and C_s = C_r = 0.1.$

The feasibility of the task set is first checked (for minimum tolerance) by using lemma 4. The run time of each task is estimated as $run_1(k_{min}) = 13.13$, $run_2(k_{min}) = 26.26$ and $run_3(k_{min}) = 39.39$, they all are less than their $min(d_i,p_i)$, hence acceptable quality is attainable.

When all tasks are released in phase i.e., at time $t = 0$, the tolerance of tasks $\tau_1, \tau_2$ and $\tau_3$ is improved up to 14, 13, 13 respectively. The run time for each is estimated as 20.18, 39.89 and 119.67. However, the subsequent release $\tau_3$ will be infeasible with this improved tolerance and can guarantee lower tolerance only. For Kmin_Kmax_Pattern, $\tau_1, \tau_2$ (released between 119.676 to 159.06) and $\tau_3$ will be scheduled with minimum required tolerance. The schedule for the task set with Kmin_Kmax_Pattern is shown in figure 4. The pattern suggested in this paper are generated using fixed priority assignment (rate-monotonic and deadline monotonic) for which the schedule can be seen from the table 3.

It can be observed from table 3 that performance of the Kmin_Pattern is worst (tolerance of 2) even after improvement. However, the performance of the deadline monotonic based patterns (Kmin_Kmax_Pattern and K_Pattern) is better than the rate monotonic based patterns (RM_Kmin_Kmax_Pattern and RM_K_Pattern). The deadline monotonic based Kmin_Kmax_Pattern or K_Pattern could provide some releases with better tolerance (14, 13, 13) than the best tolerance given to any release by rate monotonic based patterns (6, 6, 6). The tolerance of the schedule can be improved by controlling preemption and utilizing the idle slots (if any).

**Phase-2: Preemption Controlled Improvement Phase**

The patterns generated in phase-1 for tasks set T allowe preemption. Decreasing number of times a lower priority release is preempted would save losses due to preemption overhead. Thus, time saved by reducing preemption can be used to increase the tolerance of the releases previously affected by this overhead. Further, overall tolerance of the system can be improved by utilizing the idle slots available.

The higher priority preempting releases can be delayed up to its laxity, so that they do not miss their deadline. This laxity can be estimated as follows: $laxity^h_i = D_i^h - T_{rel^h_i}^k(k_i,m_i)\forall k_i \in H_{(i,j)}$ where $H_{(ij)}$ is the set of jobs preempting $\tau_i^h$ such that $Rel^h_i > Rel^h_j$. Hence, time available for execution non-preemptively by the lower priority release would be $Ta^h_i = \min\{min_{k_i \in H_{(i,j)}}(laxity^h_i)_{\tau_i^h < cur_t}, (Rel^h_i + laxity^h_i), (D_i^h - Rel^h_i)\}$ where $t_{cur_t}$ is the current time of the system, when no higher priority release is available then $rel^h_i = \infty$. If the time available is sufficient to complete the release $\tau_i^h$ non-preemptively then it is allowed to do so. The overhead thus, saved can be used by the lower priority release to improve its tolerance. In case, the tolerance of lower priority release is already maximum or overhead saved is insufficient to further improve the tolerance the subsequent release is given a chance and so on. However, in case the time available is insufficient to complete release $\tau_i^h$ then $\tau_i^h$ is executed non-preemptively for $Ta^h_i$ and then preempted with the overhead. Further, the idle slots available are used by the release finishing just before it to improve its tolerance. In case, its tolerance is already maximum the previous release is given a chance and so on.

For the example 2 (table 3), if the preemption overhead is $\delta = 0.4$. In such a case, the maximum tolerance, that can be given to any release of $\tau_1, \tau_2$ and $\tau_3$ will be 13 tolerance. Then the finish time $ft^h_3(k_{max}) = 119.16$. Refer figure 5a. The second release of $\tau_3$ would guarantee tolerance to only minimum $k_{min} = 2$ faults. However, when preemption control technique is applied in the phase-2 to this schedule, the release $\tau_3^h$ will be able to finish at 59.13 and relieve the system from the preemption overhead of 0.9 while honoring the deadline of the higher priority releases. Since, tolerance of $\tau_3^h$ is 13 which is maximum. The overhead time saved (0.9) is not used by $\tau_3^h$, rather its subsequent release is given chance for improvement (refer figure 5b). In this case tolerance of $\tau_3^h$, which was originally 2 can be improved to 3. And the subsequent release is preponed by 0.36 units of time, which may be used to improve its tolerance.

**Improved Acceptability Domain**

**Example 3:** Consider a task set $T = \{ \tau_1, \tau_2 \} = \{ (e_1, p_1, d_1, k_{min}^1): (10,25,50,1), (15,5,50,65,1) \}$ and $C_s = C_r = 0.4,$ $\delta = 0.4$.

The requirement time to tolerate minimum number of faults is 14.4 and 20.9 for tasks $\tau_1$ and $\tau_2$ respectively. The finish time of the critical instance release i.e., $\tau_1^h$ and $\tau_2^h$ will be 14.4 and 64.9 respectively. However, the second release $\tau_2^h$ with release time $Rel_2 = 50$ and absolute deadline of $D_2 = 115$, will finish at 115.4 and hence, infeasible, in case it will tolerate $k_{min} = 1$ fault. Thus, this task set is infeasible, since minimum acceptable quality cannot be achieved within deadline (figure 6a).

**Burst tolerance for multiple task**

The response time of a task depends on the other task in the task set. Hence, burst tolerance of a release $\tau_i^h$ will depend on:

- **a.** the number of checkpoints deployed in this release. Thus, if $b^h_i$ faults occur in $\tau_i^h$ release of $\tau_i$ then worst case requirement time $B^h_i(m_i^h, b^h_i) = \xi_i(m_i) + b^h_i \cdot r_{b^h_i}(m_i^h)$..(14)

- **b.** the time required by the higher priority task, released during the run time of this release (i.e., $Rel^h_i, B^h_i$). Thus, $\{ t_h^h \tau_{h+1}^h \tau_{h+2}^h \ldots \tau_{h+h^*}\text{no_high}^h \}$ releases will preempt $\tau_i^h$ during $(Rel^h_i, B^h_i)$ where $h = 1, 2, \ldots i - 1, \text{for } h = [Rel^h_i/p_i]$ and no_high$^h = [D_i^h/p_i] - [Rel^h_i/p_i].$ If $b_i^h$ faults occur during the $\tau_h^h$ release then it will require $B^h_i(m_i^h, b_i^h)$ amount of
time (equation (14)). For all the no_high releases the required time will be $\sum \mathcal{B}(m^h, b^h)$. Hence the total time required by all the higher priority releases will be

$$req_{\text{high}} = \sum_{h=1}^{i-1} \sum_{j=1}^{f_{h, \text{no, high}}} \mathcal{B}(m^h, k_{\text{min}})$$

(15)

c. the laxity of the lower priority task release $\tau^i_t$. If task $\tau^j_t$ is preempted by $\tau^i_t$ (i.e., $(\text{Rel}^j_t \leq \text{Rel}^i_t)$) then its response time will affect the response time of $\tau^i_t$ which has to finish by its deadline $(\text{Rel}^i_t, D^i_t)$. But in case there is no lower priority release available in the waiting queue (i.e., $(\text{Rel}^j_t < \text{Rel}^i_t < D^j_t)$) then $\tau^i_t$ can avail the laxity of the upcoming lower priority release $(\text{Rel}^i_t, D^i_t)$ and can have an additional time of
Figure 5a: Timing Diagram for task set T (example 2) using Kmin_Kmax_Pattern after phase-1

Indicates the release time
Indicates the finish time
Indicates execution of a release

Figure 5b: Timing Diagram for the task set T (example 2) using Kmin_Kmax_Pattern after phase-2

Indicates idle slot
Indicates infeasible release
Indicates execution of a release

Figure 6a: Task set infeasible at accepted QoS. Release r12 misses deadline of 115

Figure 6b: Feasible schedule after phase-2
\((Rel_i^j - Rel_i^l)\). Thus, the window for consideration will be \((Rel_i^l, D_i^l)\).

The burst tolerance of the \(j\)th release of task \(r_j\) can be predicted by the function \(burst\_tol\_task_j\) given Appendix 2.

The burst tolerance measurement of each task for all the five patterns can be seen in the table 3. The results are simulated by assuming that \(j^{th}\) release of task \(r_i\) can tolerate a burst of \(bt_i^j\) when all other releases had actually faced minimum number of faults. The burst tolerance of the task set is the minimum burst tolerance of all the releases. It can be seen from the table 3 that the burst tolerance of KminPattern is lowest (i.e., 2) indicating that the system would fail in case of sudden burst of faults over a small duration of time appears.

5 Simulation Results:

Simulations for synthesized as well as real world applications are performed to evaluate performance of the proposed two phase preemption control pattern approach with respect to KminPattern [11, 33].

Results are reported for after each phase. For deadline monotonic based fault tolerance patterns after phase-1 are referred to as K Pattern and Kmin_Kmax_Pattern. If priorities are assigned based on rate monotonic scheme, they are referred to as RM_K_Pattern and RM_Kmin_Kmax_Pattern. However, after second phase, results are reported as PC_Kmax, PC_K, PC_RM_Kmax and PC_RM_K for preemption controlled Kmin_Kmax_Pattern, K_Pattern, rate monotonic RM_Kmin_Kmax_Pattern and RM_K_Pattern respectively.

Key parameters used for comparisons are average tolerance and burst tolerance defined as follows:

Average tolerance is ratio of sum of tolerances of each release to the number of releases appearing in a pattern.

Burst tolerance of a release is measured as maximum number of faults it can tolerate assuming minimum number of faults appeared in all previous releases. Thus, burst tolerance of a task set (or simply burst tolerance) is minimum burst tolerance of all releases.

5.1 Simulations with Synthesized Task Sets

All simulations are done by considering task set \(T = \{r_1, r_2, r_3 \ldots r_n\}\) with utilization \(U\) (a uniform random number in the range \((0,1)\)) on a system. Checkpoint saving cost \(C_s\) (a uniform random number in range \((0,30)\)) and retrieval cost \(C_e = C_r\). Preemption overhead \(R\) is chosen uniformly in the range \((0, 10)\). For each task \(r_i\), attributes are chosen according to table 4. The results are obtained by taking the average of 100 task sets.

Effect of Utilization \(U\) on Average Tolerance

Effect of utilization on average tolerance can be seen from figure 7a and 7b. Figure 7a compares the performance of K_Pattern, PC_K, RM_K_Pattern, PC_RM_Kmax and Kmin_Pattern. The average tolerance of Kmin_Pattern is lowest for the entire utilization range. The decrement in the tolerance for rate monotonic based patterns (RM_K_Pattern, PC_Kmax, PC_K) are assigned based on rate monotonic scheme, they are referred to as RM_K_Pattern and RM_Kmin_Kmax_Pattern.
is more than the deadline monotonic based patterns. But, for medium utilizations (0.3, 0.7) the deadline monotonic based patterns provide better average tolerance compared to the rate monotonic or the Kmin_Pattern. In case of higher utilization the free time slots available are rare hence, there is little opportunity for improvement in fault tolerance. The Kmin_Pattern could not provide any tolerance to most of the task sets considered in this utilization range while both deadline monotonic and rate monotonic based patterns could provide few tolerance to some releases, lowering their probability of rejection although all the averages are almost same.

Effect of Utilization U on Burst Tolerance

Effect of utilization on the burst tolerance can be seen from the figure 8a and 8b. It is observed from the figures that burst tolerance of Kmin_Pattern indicating that it is least resistant to unforeseen faults. However, the burst tolerance of the deadline monotonic based patterns is better than the rate monotonic based patterns.

It is observed (figure 7a, 7b, 8a and 8b) that the deadline monotonic based pattern provides better average tolerance and burst tolerance to task sets in all utilization range than the
rate monotonic based patterns. This is due to the fact that the maximum tolerance a release of a task can get depends on the slack between the period and the deadline of this task as well as that of lower priority tasks. If lower priority task has shorter deadline than this task, then to ensure feasibility of lower priority task this task is indirectly restricted by lower deadline, leading to lower tolerance.

Thus, the rest of the results are compared for the deadline monotonic based patterns only. In all the following subsections (figures 9-16) task sets are generated with utilization U = 0.5.

Effect of deadline to period ratio On Average tolerance

Effect of deadline to period ratio can be observed on the average tolerance for task sets in figure 9. It can be observed that for task sets with deadlines less than equal to period all patterns provide same tolerance. However, for deadline greater than period an improvement in the tolerance for K_Pattern is observed and still better tolerance is seen for Kmin_Kmax_Pattern over Kmin_Pattern. This is due to the fact that K_Pattern and PC_K is greedy in nature and tends to fall around a mean while Kmin_Pattern neglects the time available beyond period in such task sets. On the other hand, the Kmin_Kmax_Pattern and PC_Kmax takes full advantage of the slack available between period and deadline (to reach new highpoint of maximum tolerance) and by providing minimum tolerance to releases (incapable of maximum tolerance) can recover quickly.

Effect of deadline to period ratio On Burst Tolerance

With increment in deadline to period ratio, burst tolerance for PC_K,K_Pattern, PC_Kmax and Kmin_Kmax_Pattern increases but there is no effect on Kmin_Pattern (Figure 10). However, this rate of improvement is greater in PC_K and K_Pattern than that observed in Kmin_Kmax_Pattern or PC_Kmax. This is because PC_K and K_Pattern remain around mean and hence has smallest recovery blocks while in case of PC_Kmax and Kmin_Kmax_Pattern recovery blocks are large when a release guarantees minimum tolerance. But after a threshold (figure 9 and 10), system tends to saturate as more and more releases are forced to wait.

Effect of minimum tolerance requirement On Average tolerance

Various systems may have different minimum tolerance requirement, if same task sets are scheduled on each of them then average tolerance of Kmin_Pattern, PC_K and K_Pattern remain indifferent while that of the Kmin_Kmax_Pattern and PC_Kmax reduces to the tolerance granted by K_Pattern (figure 11). This is due to the fact that for Kmin_Kmax_Pattern or PC_Kmax, an increment in k_min reduces the relaxation slot of A while the overload slot δ is constant (figure 3) hence more releases are required to be at minimum tolerance. In case of Kmin_Pattern, PC_K and K_Pattern this slot is constant.

Effect of minimum tolerance requirement On Burst Tolerance

With increment in minimum tolerance requirement PC_Kmax and Kmin_Kmax_Pattern eventually are same as PC_K and K_Pattern respectively (Figure 12). This is because as minimum requirement increases recovery block size decreases and hence the burst tolerance increases for PC_Kmax and Kmin_Kmax_Pattern. But PC_K, K_Pattern and Kmin_Pattern are independent of minimum tolerance.
Effect of checkpoint saving cost to average execution time ratio on average tolerance

Effect of checkpoint overheads on average tolerance is shown in figure 13. The performance can be obtained by considering two cases, one in which the task is greedy to apply checkpoint invariably (greedy K_Pattern) while the other (K_Pattern) apply only when it is beneficial as discussed in lemma 2. As checkpoint saving cost to average execution time ratio \( \text{ratio_cs_e} \) increases checkpoint deployment cost increases, hence average tolerance decreases for greedy_K_Pattern. On other hand, as \( \text{ratio_cs_e} \) increases K_Pattern discontinue checkpoint application and maintain a constant rollback equal to execution time. Similar trend is noticed for Kmin_Kmax_Pattern.

Effect of checkpoint saving cost to average execution time ratio on burst tolerance

For the burst tolerance, performance of K_Pattern will be similar to that observed for average tolerance because as the \( \text{ratio_cs_e} \) increases the K_Pattern do not employ checkpointing hence, a constant burst tolerance would be observed for the rollback equal to the execution time (refer figure 14). However, for greedy_K_Pattern the burst tolerance first decreases and then becomes equal to the constant tolerance provided by the K_Pattern because till it keeps applying checkpoints for higher ratios (0.3 in figure 14) the influence of checkpoint is negative but once it reaches zero (i.e., to no tolerance at 0.9) then it also stops applying checkpointing, then the two schemes would be equivalent.

Effect of \( \text{ratio_po_e} \) on average tolerance

Effect of increment in preemption overhead ratio on average tolerance is shown in figure 15. The average tolerance of all patterns reduces with increment in the ratio because higher preemption overhead reduces time available for a release to tolerate faults. Performance of the Kmin_Kmax_Pattern is better than of K_Pattern because response time of most of the releases in Kmin_Kmax_Pattern is shorter (execute at minimum tolerance) as compared to those in K_Pattern. Thus, increment in preemption overhead has a profound effect on K_Pattern. However, both PC_K and PC_Kmax control preemption, hence, provide better average tolerance. When preemption overhead ratio is substantially high then Kmin_Pattern, K_Pattern and Kmin_Kmax_Pattern may fail to provide acceptable quality and hence, will not accept such task sets. While PC_K and PC_Kmax patterns perform preemption control and can accept more task sets.

5.2 Simulations with Real World Applications

Benchmarked real world task sets, i.e., Computerized Numerical Control (CNC) [34] (consisting of 8 tasks, execution time range (0.035-0.72ms), period range (2.4-9.6ms)), Inertial Navigation System (INS) [35, 36] (consisting of 6 tasks, execution time range (1.18-2.5ms), period range (2.5-1250ms)), Generic Aviation Platform (GAP) [37] (17 tasks, execution time range (1-9ms), period range (5-1000ms)) and Webphone (WP) [38] (4 tasks, execution time range (1.383-50.386s), period range of (40-66.675s)). All simulations are done on Transmeta Crusoe (300MHz) [11]. The checkpoint and preemption context size is assumed to be 5kB and saved in DRAM. The checkpoint and preemption overheads are assumed to be 0.4ms [11, 48]. Average and burst tolerance for each is shown in figure 17 and 18 respectively. Real-world applications show similar trends as synthesized task sets. Average and burst tolerance of CNC machine is high due to lower utilization. However, video encoding task in webphone application places a high computational requirement leading to lower improvement in average and burst tolerance.
6 Conclusion

This paper proposes a two phase preemption control pattern approach for periodic task sets with arbitrary deadlines by utilizing the concept of checkpointing and preemption control. Lemmas (1 and 2) are used to decide the number of checkpts whereas condition for preemption control is estimated to allow the lower priority release to complete earlier. Lemma (3 and 4) and theorem (1 and 2) are derived to ensure the feasibility of the task set with acceptable quality. To improve the average quality of service fault tolerance patterns such as, minimum and maximum tolerance pattern (Kmin_Kmax_Pattern) and greedy pattern (K_Pattern) have been proposed taking into account the preemption overhead incurred due to context switching between releases. Further, the tolerance of the system is improved by preemption control in the second phase by PC_K and PC_Kmax. As demonstrated from the example, some task sets that were rejected by the existing techniques are also accepted. The burst tolerance measurement of each pattern is done to predict its behavior in case of unforeseen faults. The effectiveness of the proposed algorithm has been discussed through examples and extensive simulation results.

It is observed that Kmin_Kmax_Pattern and PC_Kmax repeat very frequently whereas average tolerance is provided to each release in K_Pattern or PC_K. The suggested patterns are applicable to the deadline monotonic as well as rate monotonic priority assignment techniques. However, the simulation results reveal that the performance of the deadline monotonic based patterns is better than the rate monotonic based patterns (approximately 18%) and approximately 53% better than the previously proposed Kmin_Pattern [11]. For different values of deadline to period ratio approximately 15% improvement is received in average tolerance of K_Pattern as compared to K_Pattern. However, it can be as high as 62% in case of Kmin_Kmax_Pattern and 65% for PC_Kmax. When the system minimum tolerance requirement is low (0,1) around 32% improvement in average tolerance is perceived in PC_Kmax. On the other hand, the burst tolerance of PC_K is approximately six times that of the Kmin_Kmax_Pattern. The K_Pattern and PC_K however, is independent of the task minimum requirement while the behavior of Kmin_Kmax_Pattern and PC_Kmax is governed by it. The proposed patterns have improvement (35% approximately) when the ratio of preemption overhead to execution time is between (0-0.8). However, for higher ratio the improvement the PC_K and PC_Kmax perform best.

Real world applications such as CNC, INS, GAP, webphone, demonstrate similar trends as received for synthesized task sets. The improvement in average tolerance of such applications is approximately 30%.

Besides providing better quality of service in terms of average tolerance and improving the possibility of survivability in case of unforeseen faults. The proposed two phase preemption control pattern approach improved the acceptability domain by accepting arbitrary deadline tasks and some previously rejected task sets with improved average tolerance.

7 Appendix

7.1 Appendix 1

Theorem 1: A uniprocessor system comprising of a single task with attributes e, p, d, kmin, if its jth release finishes by its absolute deadline Dj then the next release will be schedulable with at least kmin tolerance.

Proof: The jth release of a task τi will be available after its release time of Retj = j * pi
Its absolute deadline is Dj = j * pi + di

While for the (j + 1)th release will arrive at
Relj+1 = (j + 1)pi

The absolute deadline would be Dj+1 = (j + 1)pi + di

If the jth release of a task finishes by its deadline (in worst case on its deadline, precisely), the wait time of the (j + 1)th release will be

Substituting equations (2) and (3) in (5) we get

Total time available for execution of (j + 1)th release will be

But in case d_i > p_i the time available for execution for (j + 1)th release can be given by substituting equations (4), (3) and (6) in (7) we get

Thus, combining equations (9) and (10) we get

Therefore, Ta[j+1] ≥ min(d_i, p_i)
From lemma3, R((kmin, mmin)) ≤ min(d, p) hence (j + 1)th release can be scheduled in the available time (11)

Theorem 2: For a given periodic task set T = {τ1, τ2, τ3, ..., τn} if the jth release of task τ_i becomes infeasible when higher priority tasks are guaranteed for number of faults (may be more than minimum) this release will be feasible if all higher priority task preempting it and itself are reduced to minimum tolerance.
Proof: For a task \( \tau_i \in T \) the finish time of \( j^{th} \) release is \((12)\)
\[
ft_i^j = \text{rel}_i^j + \text{run}_i^j(k_i, m_i) + \left( \max(0, \text{df}_i^{j-1} - \text{rel}_i^j) \right)
\]
Proving by induction method.

For the first release of \( \tau_m(\tau_n^0) \)
\[
ft_0^0 = 0 + \text{run}_m^0(k_m, m_0) + 0 \leq D_0^0.
\]
Now, we have to prove if the \((j-1)^{th}\) release is feasible then \( j^{th} \) release is also feasible with minimum tolerance \( k_{min} \).

Consider \((j-1)^{th}\) release is feasible
\[
\Rightarrow ft_i^{j-1} \leq D_i^{j-1}. \text{In worst case we assume, } ft_i^{j-1} = D_i^{j-1}.
\]
\[
ft_i^j = \text{rel}_i^j + \text{run}_i^j(k_{min}, m_{min}) + \left( \max(0, \text{df}_i^{j-1} - \text{rel}_i^j) \right)
\]
If \( D_i^{j-1} > \text{rel}_i^j \) then \( ft_i^j = \text{run}_i^j(k_{min}, m_{min}) + D_i^{j-1} \)
If \( D_i^{j-1} \leq \text{rel}_i^j \) then \( ft_i^j = \text{rel}_i^j + \text{run}_i^j(k_{min}, m_{min}) \)
Combining the above equations we get
\[
ft_i^n = \text{run}_i^n(k_{min}, m_{min}) + \max(D_i^{j-1}, \text{rel}_i^j). \text{ For } j^{th} \text{ release time available for execution will be } T_a_i^j = D_i^j - \max(D_i^{j-1}, \text{rel}_i^j) \geq \min(d_i, p_i) \text{ by theorem1.}
\]
Substituting \( \max(D_i^{j-1}, \text{rel}_i^j) \) in \( ft_i^n \) we get
\[
ft_i^n = \text{run}_i^n(k_{min}, m_{min}) + D_i^j - T_a_i^j
\]
\[
\Rightarrow D_i^j = ft_i^n + T_a_i^j - \text{run}_i^n(k_{min}, m_{min}).
\]
By lemma4 \( \text{run}_i^n(k_{min}, m_{min}) \leq \min(d_i, p_i) \) for critical release \( \tau_n \).
Thus, \( \text{run}_i^n(k_{min}, m_{min}) \leq \text{run}_i^n(k_{min}, m_{min}) \).
Hence, \( ft_i^n \leq D_i^n \).
This implies that each release of task \( \tau_i \in T \) will be feasible with at least minimum tolerance in worst case. Hence, the task set \( T \) will be feasible. \(\blacksquare\)

### 7.2 Appendix 2

**Function calculate_kmax(e_i, p_i, d_i, k_{min_i})**

// Function to estimate maximum tolerance \( k_{max_i} \) of a task \( \tau_i \),
//while it satisfies the minimum tolerance requirement \( k_{min_i} \)

**Begin**

1. Initially \( k_{max_i} = k_{min_i} \)
2. \( R_i(k_{max_i}, m_{max_i}) = \text{Calculate\_Requirement\_Time(e_i, p_i, d_i, k_{max_i})} \)
3. While \( (R_i(k_{max_i}, m_{max_i}) \leq d_i) \& \& k_{max_i} \leq k_{obs_i} \)
   Do
   a. \( k_{max_i} = k_{max_i} + 1 \)
   b. \( R_i(k_{max_i}, m_{max_i}) = \text{Calculate\_Requirement\_Time(e_i, p_i, d_i, k_{max_i})} \)
   Repeat
4. \( k_{max_i} = k_{max_i} - 1 \)
5. Return \( k_{max_i} \)
**End**

Computation requirement with \( k_{max_i} \) tolerance will be \( R_i(k_{max_i}, m_{max_i}) \leq d_i \). Then best quality of service equivalent tok_{max_i} tolerance can be provided, if the deadline of this task is less than or equal to its period(\( d_i \leq p_i \)). This is because, as suggested by lemma 3, utilization \( u_i \) would be \( R_i(k_{max_i}, m_{max_i})/d_i \) which is always less than or equal to one. However, if both deadline and required time are greater than period (i.e., \( d_i > p_i \)) and \( p_i < R_i(k_{max_i}, m_{max_i})/d_i \) then utilization \( u_i \) would be \( R_i(k_{max_i}, m_{max_i})/p_i \) which is always greater than one, thus all releases cannot be granted \( k_{max_i} \) tolerance, leading to a variable tolerance in the range of \( [k_{min_i}, k_{max_i}] \).

### 8 References


