On the Design of Contention Managers for Real-Time Software Transactional Memory

Mohammed Elshambakey

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Robert P. Broadwater
Cameron D. Patterson
Mohamed Rizk Mohamed. Rizk
Anil Kumar S. Vullikanti

April 10, 2012
Blacksburg, Virginia

Keywords: Software Transactional Memory, Embedded Systems, Contention Managers

On the Design of Contention Managers for Real-Time Software Transactional Memory

Mohammed Elshambakey

(ABSTRACT)

Real-time systems is concerned with time constraints imposed on a set of tasks executed on one or more processors. Each task consists of a set of instances (jobs). Each job can be repeated exactly at fixed time interval (periodic), or at least at fixed time interval (sporadic). Each job has an absolute deadline. Hard real-time systems (HRT) do not allow jobs to exceed their absolute deadline, while soft real-time systems (SRT) permit exceeding absolute deadlines within upper bound. Real-time schedulers try to satisfy deadline constraints for HRT, and bounding tardiness (the amount of time that a job exceeds its absolute deadline) for SRT. Global Earliest Deadline First (G-EDF) and Global Rate Monotonic Assignment (G-RMA) are two examples of multiprocessor real-time schedulers. G-EDF gives a higher priority to the instance of earliest absolute deadline, while G-RMA gives higher priority to instance of smallest period.

Tasks may need to access common objects through read/write operations. Different instances can conflict if they try to access the same object, and at least one instance is writing to it. Thus, in addition to real-time scheduling, concurrency control is needed for real-time systems. Concurrency control for real-time system should consider time constraints. A number of real-time concurrency control methods were introduced based on locking, lock-free and wait-free algorithms. Lock-based concurrency control suffers from programmability, scalability, and compositionality challenges. These challenges are exacerbated in emerging multicore architectures, on which improved software performance must be achieved by exposing greater concurrency. Lock-free and wait-free offer numerous advantages over locks (e.g., deadlock-freedom), but their programmability has remained a challenge. Past studies show that they are best suited for simple data structures where their retry cost is competitive to the cost of lock-based synchronization.

Transactional memory (TM) is an alternative synchronization model for shared in-memory data objects that promises to alleviate difficulties in locking and lock-free algorithms. With TM, programmers write concurrent code using threads, but organize code that read/write shared objects as transactions, which appear to execute atomically. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager (or CM) resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. The time consumed by a transaction in abortion and retrying is known as retry cost. In addition to a simple programming model, TM provides performance comparable to highly concurrent fine-grained locking and lock-free approaches, and is composable. TM is semantically simpler and is often the only viable lock-free solution for complex data structures and nested critical sections. Multiprocessor TM has been proposed in hardware, called HTM, and in software, called STM, with the usual tradeoffs: HTM provides strong atomicity, has lesser overhead, but needs transactional support in hardware; STM is available on any hardware.

In this dissertation proposal, we consider STM for concurrency control in multicore real-time software. Doing so will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy time constraints. Retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are scheduler-dependent). Thus, real-time CM is logical.

We investigate and design a number of real-time CMs. The first two CMs are directly based on dynamic and static priority of underlying tasks. Earliest Deadline-First CM with G-EDF scheduler (ECM) resolves conflicts based on absolute deadline of the underlying instances. Rate Monotonic Assignment with G-RMA scheduler (RCM) resolves conflicts based on period of underlying instances. We analyze retry cost and response time under ECM and RCM. We analytically and experimentally compare their schedulability against lock-free method.

ECM and RCM conserve the semantics of the underlying real-time scheduler. This conservative approach results in a maximum retry cost- for a single transaction due to another transaction- of double the maximum atomic section length among all tasks. So, another CM is developed to reduce this retry cost. Length-based CM (LCM) considers not only static/dynamic priority of underlying instance, but also length of the interfering transaction compared to remaining length of interfered transaction. LCM is used with G-EDF and G-RMA. Although it can reduce retry cost, but it suffers from priority inversion. By proper choice of different parameters, additional cost due to priority inversion can be kept lower than reduced retry cost. Thus, the net result will be lower response time for tasks using LCM with G-EDF/G-RMA. We analyze retry cost and response time of LCM. We analytically and experimentally compare LCM schedulability against ECM, RCM and lock-free.

ECM, RCM and LCM are affected by transitive retry. Transitive retry enforces a transaction to abort and retry due to another non-conflicting transaction. Transitive retry appears when multiple objects exist per transaction. So, we develop the Priority-based with Negative value and First access (PNF) contention manager. PNF avoids transitive retry and deals better with multiple objects than previous contention managers. PNF also tries to optimize processor usage by lowering priority of the job underlying retrying transaction. Thus, other jobs can proceed if there is no conflict. We upper bound retry cost and response time for PNF when used with G-EDF and G-RMA. Schedulability is compared between PNF on one side and ECM, RCM, LCM and lock-free on the other. We experimentally compare retry cost of PNF compared to other synchronization techniques.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Data structure implementation affects degree of exploited parallilsm in the system [71]. As the implementation moves from lock-based to non-blocking, more parallilism can be exploited. Thus, more computation speed-up, but more complex design. Things become more complicated when it comes to synchronization under real-time multiprocessor systems. Real-time systems should meet deadline of each task as much as possible. With shared objects, tasks have to execute in a serial order to some degree when accessing shared objects. The major problem arises when one of the accesses is a *write* operation. Conflict resultion techniques is used to decide how to access these shared objects. In lock-based systems, tasks try to obtain an object's lock before accessing it. While in lock-free techniques, some intuition is used to allow tasks to simultaneously access the object without violating consistency. Different synchronization techniqes affect response time of each task. Depending on the rationale of conflict resolution, a task might have to wait for all other interfering tasks before it can access the required object.

1.0.1 Preliminaries

We consider a multiprocessor system with m identical processors and n sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$. The k^{th} instance (or job) of a task τ_i is denoted τ_i^k. Each task τ_i is specified by its worst case execution time (WCET) c_i, its minimum period T_i between any two consecutive instances, and its relative deadline D_i, where $D_i = T_i$. Job τ_i^j is released at time r_i^j and must finish no later than its absolute deadline $d_i^j = r_i^j + D_i$. Under a fixed priority scheduler such as G-RMA, p_i determines τ_i's (fixed) priority and it is constant for all instances of τ_i. Under a dynamic priority scheduler such as G-EDF, τ_i^j's priority, p_i^j, is determined by its absolute deadline. A task τ_j may interfere with task τ_i for a number of times during a duration L, and this number is denoted as $G_{ij}(L)$. τ_j's workload that interferes with τ_i during L is denoted $W_{ij}(L)$.

Shared objects. A task may need to access (i.e., read, write) shared, in-memory objects while it is executing any of its atomic sections, which are synchronized using STM. The set of atomic sections of task τ_i is denoted s_i. s_i^k is the k^{th} atomic section of τ_i. Each object, θ, can be accessed by multiple tasks. The set of distinct objects accessed by τ_i is θ_i. The set of atomic sections used by τ_i to access θ is $s_i(\theta)$, and the sum of the lengths of those atomic sections is $len(s_i(\theta))$. $s_i^k(\theta)$ is the k^{th} atomic section of τ_i that accesses θ. $s_i^k(\theta)$ executes for a duration $len(s_i^k(\theta))$.

If θ is shared by multiple tasks, then $s(\theta)$ is the set of atomic sections of all tasks accessing θ, and the set of tasks sharing θ with τ_i is denoted $\gamma_i(\theta)$. Atomic sections are non-nested. Each atomic section is assumed to access only one object; this allows a head-to-head comparison with lock-free synchronization [25]. (Allowing multiple object access per transaction is future work.) The maximum-length atomic section in τ_i that accesses θ is denoted $s_{i_{max}}(\theta)$, while the maximum one among all tasks is $s_{max}(\theta)$, and the maximum one among tasks with priorities lower than that of τ_i is $s_{max}^i(\theta)$.

STM retry cost. If two or more atomic sections conflict, the CM will commit one section and abort and retry the others, increasing the time to execute the aborted sections. The increased time that an atomic section $s_i^p(\theta)$ will take to execute due to interference with another section $s_j^k(\theta)$, is denoted $W_i^p(s_j^k(\theta))$. The total time that a task τ_i's atomic sections have to retry is denoted $RC(\tau_i)$. When this retry cost is calculated over the task period T_i or an interval L, it is denoted, respectively, as $RC(T_i)$ and $RC(L)$.

1.0.2 Summary of Current Research and Contributions

Contribution of the proposal can be summarized as follows:-

- We investigate and design priority-based contention managers for real-time systems. These contention managers try to preserve time constraints in addition to data accuracy. For this goal, we investigate Earliest Deadline First contention manager (ECM) and present Rate Monotonic Assignment contention manager (RCM). ECM and RCM keeps the logic of the underlying real-time scheduler (i.e., transaction belonging to higher priority job is allowed to commit first).

- We present Length-based contention manager (LCM) which can be used with both G-EDF and G-RMA. LCM is not only concerned with priority of the transactions, but also with the length of the interfering transaction relative to the length of the interfered transaction. LCM achieves better retry cost and response time than ECM and RCM.

- Priority-based with Negative value and First access (PNF) contention manager is introduced. PNF avoids transitive retry effect suffered by ECM, RCM and LCM in case of multiple objects per transaction. PNF tries to optimize processor usage by lower priority of aborted transaction. This way, other tasks can proceed if they do no conflict with others.

- For previous contention managers, we upper bounded their retry cost and response times. We compared their schedulability to identify the conditions to prefer one of the them over the others. We also compared their schedulability against schedulability of lock-free method. We also compared retry cost of previous synchronization techniques.

1.0.3 Summary of Proposed Post Preliminary-Exam Work

Based on our current research results, we proposed the following work:

- **Analytical and experimental comparison between developed CMs and real-time locking protocols** It has been said that lock-free and wait-free methods offer numerous advantages over locking protocols, but locking protocols are still of wide use in real-time systems due to simpler programming and analysis than lock-free. Thus, it is desired to compares different CMs against real-time locking protocols. Examples of real-time locking protocols include PCP and its variants [18, 44, 62, 70], multiprocessor PCP (MPCP) [14, 28, 46, 61], SRP [50], multiprocessor SRP (MSRP) [35], PIP [28], FMLP [10,11,43] and OMLP [7]. OMLP and FMLP are similar, and FMLP was found to be superior to other protocols [13].

- **Contention manager development for nested transactions** Transactions can be nested *linearly*, where each transaction has at most one pending transaction [58]. Nesting can also be done in *parallel* where transactions execute concurrently within the same parent [77]. Linear nesting can be *1) flat:* If a child transaction aborts, then parent also aborts. If a child commits, no effect is taken until the parent commits. Modifications made by child transaction is seen only be the parent . *2) Closed:* Similar to *flat nesting* except that if a child transaction aborts, parent does not have to abort. *3) Open:* If a child transaction commits, its modifications is seen not only by the parent, but also by other non-surrounding transactions. If parent aborts after child commits, child modifications are still valid. It is required to extend the proposed real-time CMs (or develop new ones) to handle some or all types of transaction nesting.

- **Combine both LCM and PNF** LCM is designed to reduce the retry cost of one transaction when it is interfered close to its end of execution. PNF is designed to avoid transitive retry in case of multiple objects per transactions. One goal is to combine benefits of both algorithms.

- **Investigate other criterion for contention managers to further reduced retry cost** Criterion other than or combined with priority, transaction length and first access may be used to produce better contention managers.

1.0.4 Proposal outline

The rest of this dissertation proposal is organized as follows. Chapter 2 overviews past and related work for real-time concurrency control. Chapter 3 investigates Earliest Deadline First CM (ECN) and proposes Rate-Monotnic Assignment CM (RCM). We derive upper bounds for retry cost and response time under ECM and RCM. Finally, schedulability is compared between ECM, RCM and lock-free method. Chapter 4 shows how to reduce retry cost of transactions under ECM and RCM using a length-based CM (LCM). Chapter 5 tries to solve transitive retry of transaction under ECM, RCM and LCM in case of multiobjects per transaction. Chapter 6 compares measured retry cost and response time for sets of tasks under previous CMs, as well as, lock-free algorithm. We conclude in Chapter 7.

Chapter 2

Past and Related Work

2.0.5 Real-Time Lock-free Synchronization

Transactional-like concurrency control and lock-free synchronization, for real-time systems, has been previously studied in (e.g., [2–5, 16, 17, 76]). Despite their numerous advantages over locks (e.g., deadlock-freedom), their programmability has remained a challenge. Past studies show that they are best suited for simple data structures where their retry cost is competitive to the cost of lock-based synchronization [12, 25]. In contrast, STM is semantically simpler [41], and is often the only viable lock-free solution for complex data structures (e.g., red/black tree) [32] and nested critical sections [1, 60, 65].

2.0.6 Transactional Memory: Overview

Transactional memory (TM) is motivated by Database transactions [36]. In TM, each thread executes a set of transactions when accessing shared memory. A TM transaction consists of a sequence of steps (i.e., reads and/or writes) executed atomically by a thread [39]. Atomicity means that the sequence of steps logically occur at a single instant in time; intermediate states are invisible to other transactions. The difficulty of locks' maintenance and development are the driving motivation for seeking alternate concurrency control methods. Lock-free and wait-free are two alternatives. Lock-free and wait-free have high performance, but significantly complex to write and reason about, and therefore, have largely been limited to a simple data structures - e.g., linked lists, queues, stacks [19–22].

The term "transactional memory" was proposed by Herlihy and Moss [42], where they presented hardware support for lock-free data structures. TM has been provided in hardware (HTM) [37, 42, 55, 59, 64], software (STM) [21, 26, 27, 38, 40, 53, 66, 72, 79] and hybrid TM [23, 45, 57, 73]. Hybrid TM allows STM to improve performance using HTM support. Conflicts between TM threads arise when multiple threads try to access the same object simultaneously, and at least one access is a *write* operation. TM uses *Contention Managers* (CM) to resolve these conflicts. CM decides which transaction to abort and when to restart the aborted transaction in case of conflicts [9, 51, 68, 74].

2.0.7 Real-Time Database Transactions

As database transactions motivated TM [36]. Real-time database has a lot of inspiration to real-time transactional memory [63]. Real-time database is not concerned only with consistency, but also with timing constraints. When there is a conflict, lower priority transaction is aborted if it is abortable, or may cause excessive blocking to any higher priority transaction [47–49]. Blocking time can be estimated by on-line or off-line schedulers. [49] and [48] proposed a framework for trading abort cost with the blocking cost of transactions. [78] and [15] present a number of transaction scheduling strategies. These strategies include ED (Earliest Deadline), HV (Highest Value), HRU (Highest Reward and Urgency), and FHR (Flexible High Reward). [75] schedules transactions on multiprocessor system based on both slack time and value assigned to each transaction. So, it tries to acheive the highest possible value of completed transactions and meat as much deadlines as possible. [80] combines EDF with LSF (Least Slack First) to compute transaction priorities. Different scheduling strategies compute transactions' priorities, hence, which transaction to commit first.

2.0.8 Real-Time STM

STM concurrency control for real-time systems has been previously studied in [6, 32–34, 52, 67, 69]. [52] proposes a restricted version of STM for uniprocessors. [34] considers STM for distributed uni-processor systems. A higher priority task causes only one retry in a lower prirority tasks due to the uni-prcessor. [33] bounds response times in distributed multiprocessor systems with STM synchronization. They consider Pfair scheduling, limit to small atomic regions with fixed size, and limit transaction execution to span at most two quanta. [67] presents real-time scheduling of transactions and serializes transactions based on transactions' - not jobs'- deadlines. However, the work does not bound retries and response times, nor establishes tradeoffs against lock-free synchronization. [69] proposes real-time HTM. The retry bound developed in [69] assumes that the worst case conflict between atomic sections of different tasks occurs when the sections are released at the same time. This assmuption deos not cover the worst case scenario for transactions' interference. [32] presents earliest-deadline CM or ECM. ECM resolves conflicts by aborting the transaction with longer absoluted deadline. [32] derives a number of properties for ECM, upper bounds transactional retrys, and compares schedulability of ECM to retry-loop lock-free synchronization [25]. [32], like [69], assumes that the worst case conflict between atomic sections occurs when the sections are released simultaneously. Besides, [32] assumes all transactions have equal lengths. [6] presents extend idea in [32] to bound number of retries and prevent starvation. [6] presents three ideas for CMs. However, work in [6] is still in progress. Provided algorithms might not give the planned results because they are not analyzed.

Chapter 3

ECM and RCM

We consider software transactional memory (STM) for concurrency control in multicore embedded real-time software. We investigate real-time contention managers (CMs) for resolving transactional conflicts, including those based on dynamic and fixed priorities, and establish upper bounds on transactional retries and task response times. We identify the conditions under which STM (with the proposed CMs) is superior to lock-free synchronization.

3.1 G-EDF/EDF CM (ECM)

Since only one atomic section among many that share the same object can commit at any time under STM, those atomic sections execute in sequential order. A task τ_i's atomic sections are interfered by other tasks that share the same objects with τ_i. Hereafter, we will use *ECM* to refer to a multiprocessor system scheduled by G-EDF and resolves STM conflicts using the EDF CM. ECM was originally introduced in [32]. ECM will abort and retry an atomic section of τ_i, $s_i^k(\theta)$ due to a conflicting atomic section of τ_j, $s_j^l(\theta)$, if the absolute deadline of τ_j is less than or equal to the absolute deadline of τ_i. ECM behaviour is shown in Algorithm 1. [32] assumes the worst case scenario for transactional retry occurs when conflicting transactions are released simultaneously. [32] also assumes all transactions have the same length. Here, we extend the analysis in [32] to a more worse conflicting scenario and consider distinct-length transactions. We also consider lower number of conflicting instances of any job τ_j to another job τ_i.

3.1.1 G-EDF Interference and workload

The maximum number of times a task τ_j interferes with τ_i is given in [8] and is illustrated in Figure 3.1. Here, the deadline of an instance of τ_j coincides with that of τ_i, and τ_j^1 is delayed

Algorithm 1: ECM

Data: $s_i^k(\theta) \rightarrow$ interfered atomic section. $s_j^l(\theta) \rightarrow$ interfering atomic section
Result: which atomic section aborts

```
if d_i^k < d_j^l then
    s_j^l(θ) aborts;
else
    s_i^k(θ) aborts;
end
```

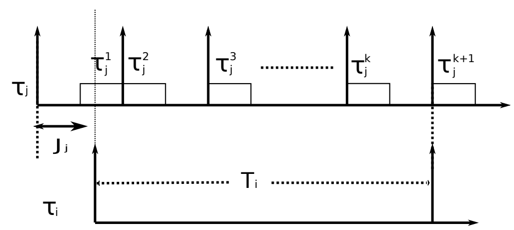


Figure 3.1: Maximum interference between two tasks, running on different processors, under G-EDF

by its maximum jitter J_j, which causes all or part of τ_j^1's execution to overlap within T_i. From Figure 3.1, it is seen that τ_j's maximum workload that interferes with τ_i (when there are no atomic sections) in T_i is:

$$\begin{aligned} W_{ij}\left(T_i\right) &\leq \left\lfloor \frac{T_i}{T_j} \right\rfloor c_j + min\left(c_j, T_i - \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) \\ &\leq \left\lceil \frac{T_i}{T_j} \right\rceil c_j \end{aligned} \tag{3.1}$$

For an interval $L < T_i$, the worst case pattern of interference is shown in Figure 3.2. Here, τ_j^1 contributes by all its c_j, and d_j^{k-1} does not have to coincide with L, as τ_j^{k-1} has a higher priority than that of τ_i. The workload of τ_j is:

$$W_{ij}\left(L\right) \leq \left(\left\lceil \frac{L - c_j}{T_j} \right\rceil + 1\right) c_j \tag{3.2}$$

Thus, the overall workload, over an interval R is:

$$W_{ij}\left(R\right) = min\left(W_{ij}\left(R\right), W_{ij}\left(T_i\right)\right) \tag{3.3}$$

where $W_{ij}(R)$ is calculated by (3.2) if $R < T_i$, otherwise, it is calculated by (3.1).

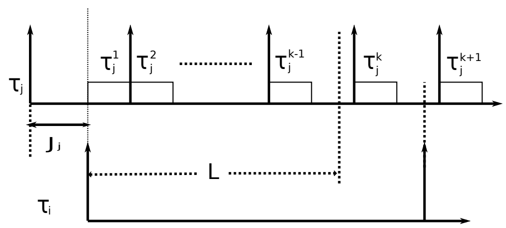
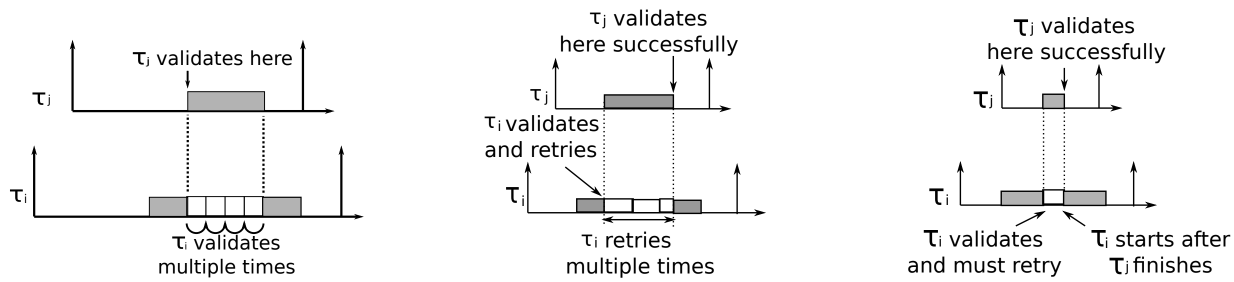


Figure 3.2: Maximum interference during an interval L of T_i



(a) Early validation

(b) Lazy validation with $len(s_i^k(\theta)) \leq len(s_j^l(\theta))$

(c) Lazy validation with $len(s_i^k(\theta)) > len(s_j^l(\theta))$

Figure 3.3: Retry of $s_i^k(\theta)$ due to $s_j^l(\theta)$

3.1.2 Retry Cost of Atomic Sections

Claim 1 *Under ECM, a task τ_i's maximum retry cost during T_i is upper bounded by:*

$$\begin{aligned} RC\left(T_i\right) \;\leq\; & \sum_{\theta \in \theta_i} \Bigg(\Bigg(\sum_{\tau_j \in \gamma_i(\theta)} \Bigg(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\big(s_j^l(\theta) \\ & + \; s_{max}(\theta)\big)\Bigg)\Bigg) - s_{max}(\theta) + s_{i_{max}}(\theta) \Bigg) \end{aligned} \tag{3.4}$$

Proof 1 *Consider two instances τ_i^a and τ_j^b, where $d_j^b \leq d_i^a$. When a shared object conflict occurs, the EDF CM will commit the atomic section of τ_j^b while aborting and retrying that of τ_i^a. Thus, an atomic section of τ_i^a, $s_i^k(\theta)$, will experience its maximum delay when it is at the end of its atomic section, and the conflicting atomic section of τ_j^b, $s_j^l(\theta)$, starts, because the whole $s_i^k(\theta)$ will be repeated after $s_j^l(\theta)$.*

Validation (i.e., conflict detection) in STM is usually done in two ways [56]: a) eager (pessimistic), in which conflicts are detected at access time, and b) lazy (optimistic), in which conflicts are detected at commit time. Despite the validation time incurred (either eager or lazy), $s_i^k(\theta)$ will retry for the same time duration, which is $len(s_j^l(\theta)+s_i^k(\theta))$. Then, $s_i^k(\theta)$ can commit successfully unless it is interferred by another conflicting atomic section, as shown in Figure 3.3.

In Figure 3.3(a), $s_j^l(\theta)$ validates at its beginning, due to early validation, and a conflict is detected. So τ_i^a retries multiple times (because at the start of each retry, τ_i^a validates) during the execution of $s_j^l(\theta)$. When τ_j^b finishes its atomic section, τ_i^a executes its atomic section.

In Figure 3.3(b), τ_i^a validates at its end (due to lazy validation), and detects a conflict with τ_j^b. Thus, it retries, and because its atomic section length is shorter than that of τ_j^b, it validates again within the execution interval of $s_j^l(\theta)$. However, the EDF CM retries it again. This process continues until τ_j^b finishes its atomic section. If τ_i^a's atomic section length is longer than that of τ_j^b, τ_i^a would have incurred the same retry time, because τ_j^b will validate when τ_i^a is retrying, and τ_i^a will retry again, as shown in Figure 3.3(c). Thus, the retry cost of $s_i^k(\theta)$ is $len(s_i^k(\theta) + s_j^l(\theta))$.

If multiple tasks interfere with τ_i^a or interfere with each other and τ_i^a (see the two interference examples in Figure 3.4), then, in each case, each atomic section of the shorter deadline tasks contributes to the delay of $s_i^p(\theta)$ by its total length, plus a retry of some atomic section in the longer deadline tasks. For example, $s_j^l(\theta)$ contributes by $len(s_j^l(\theta) + s_i^p(\theta))$ in both Figures 3.4(a) and 3.4(b). In Figure 3.4(b), $s_k^y(\theta)$ causes a retry to $s_j^l(\theta)$, and $s_h^w(\theta)$ causes a retry to $s_k^y(\theta)$.

Since we do not know in advance which atomic section will be retried due to another, we can safely assume that, each atomic section (that shares the same object with τ_i^a) in a shorter deadline task contributes by its total length, in addition to the maximum length between all atomic sections that share the same object, $len(s_{max}(\theta))$. Thus,

$$W_i^p\left(s_j^k(\theta)\right) \le len\left(s_j^k(\theta) + s_{max}(\theta)\right) \tag{3.5}$$

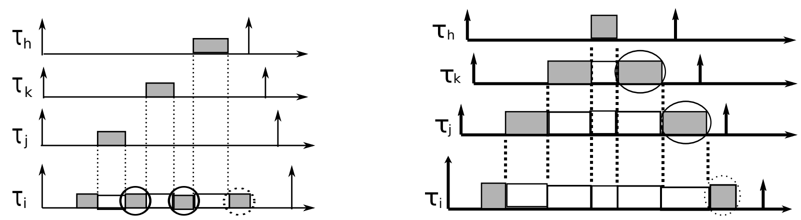
Thus, the total contribution of all atomic sections of all other tasks that share objects with a task τ_i to the retry cost of τ_i during T_i is:

$$\begin{aligned} RC(T_i) \;\le\; & \sum_{\theta\in\theta_i}\sum_{\tau_j\in\gamma_i(\theta)}\Bigg(\left\lceil\frac{T_i}{T_j}\right\rceil\sum_{\forall s_j^l(\theta)} len\big(s_j^l(\theta) \\ & + \; s_{max}(\theta)\big)\Bigg) \end{aligned} \tag{3.6}$$

Here, $\left\lceil\frac{T_i}{T_j}\right\rceil\sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta) + s_{max}(\theta)\right)$ is the contribution of all instances of τ_j during T_i. This contribution is added to all tasks. The last atomic section to execute is $s_i^p(\theta)$ (τ_i's atomic section that was delayed by conflicting atomic sections of other tasks). One of the other atomic sections (e.g., $s_m^n(\theta)$) should have a contribution $len(s_m^n(\theta) + s_{i_{max}}(\theta))$, instead of $len(s_m^n(\theta) + s_{max}(\theta))$. That is why one $s_{max}(\theta)$ should be subtracted, and $s_{i_{max}}(\theta)$ should be added (i.e., $s_{i_{max}}(\theta) - s_{max}(\theta)$). Claim follows.

Claim 2 *Claim 1's retry bound can be minimized as:*

$$RC(T_i) \le \sum_{\theta\in\theta_i} min(\Phi_1, \Phi_2) \tag{3.7}$$



(a) Other atomic sections interfere only with $s_i^p(\theta)$

(b) All atomic sections interfere with each other and $s_i^p(\theta)$

Replaced in calculations by $s_{max}(\theta)$

Replaced in calculations by $s_{i_{max}}(\theta)$

Figure 3.4: Retry of $s_i^p(\theta)$ due to other atomic sections

where Φ_1 is calculated by (3.4) for one object θ (not the sum of objects in θ_i), and

$$\Phi_2 = \Bigg(\sum_{\tau_j \in \gamma_i(\theta)} \Big(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\big(s_j^l(\theta) + s^*_{max}(\theta)\big)\Big)\Bigg) - \bar{s}_{max}(\theta) + s_{i_{max}}(\theta) \tag{3.8}$$

where s^*_{max} is the maximum atomic section between all tasks, except τ_j, accessing θ. $\bar{s}_{max}(\theta)$ is the second maximum atomic section between all tasks accessing θ.

Proof 2 *(3.4) can be modified by noting that a task τ_j's atomic section may conflict with those of other tasks, but not with τ_j. This is because, tasks are assumed to arrive sporadically, and each instance finishes before the next begins. Thus, (3.5) becomes:*

$$W_i^p\left(s_j^k(\theta)\right) \le len\left(s_j^k(\theta) + s^*_{max}(\theta)\right) \tag{3.9}$$

To see why $\bar{s}_{max}(\theta)$ is used instead of $s_{max}(\theta)$, the maximum-length atomic section of each task that accesses θ is grouped into an array, in non-increasing order of their lengths. $s_{max}(\theta)$ will be the first element of this array, and $\bar{s}_{max}(\theta)$ will be the next element, as illustrated in Figure 3.5, where the maximum atomic section of each task that accesses θ is associated with its corresponding task. According to (3.9), all tasks but τ_j will choose $s_{j_{max}}(\theta)$ as the value of $s^*_{max}(\theta)$. But when τ_j is the one whose contribution is studied, it will choose $s_{k_{max}}(\theta)$, as it is the maximum one not associated with τ_j. This way, it can be seen that the maximum value always lies between the two values $s_{jmax}(\theta)$ and $s_{kmax}(\theta)$. Of course, these two values can be equal, or the maximum value can be associated with τ_i itself, and not with any one of the interfering tasks. In the latter case, the chosen value will always be the one associated with τ_i, which still lies between the two largest values.

This means that the subtracted $s_{max}(\theta)$ in (3.4) must be replaced with one of these two values ($s_{max}(\theta)$ or $\bar{s}_{max}(\theta)$). However, since we do not know which task will interfere with τ_i, the

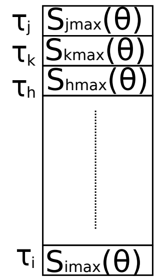


Figure 3.5: Values associated with $s^*_{max}(\theta)$

minimum is chosen, as we are determining the worst case retry cost (as this value is going to be subtracted), and this minimum is the second maximum.

Since it is not known a-priori whether Φ_1 will be smaller than Φ_2 for a specific θ, the minimum of Φ_1 and Φ_2 is taken as the worst-case contribution for θ in $RC(T_i)$.

3.1.3 Upper Bound on Response Time

To obtain an upper bound on the response time of a task τ_i, the term $RC(T_i)$ must be added to the workload of other tasks during the non-atomic execution of τ_i. But this requires modification of the WCET of each task as follows.

c_j of each interfering task τ_j should be inflated to accommodate the interference of each task τ_k, $k \neq j, i$. Meanwhile, atomic regions that access shared objects between τ_j and τ_i should not be considered in the inflation cost, because they have already been calculated in τ_i's retry cost. Thus, τ_j's inflated WCET becomes:

$$c_{ji} = c_j - \left(\sum_{\theta \in (\theta_j \wedge \theta_i)} len\left(s_j(\theta)\right) \right) + RC(T_{ji}) \tag{3.10}$$

where, c_{ji} is the new WCET of τ_j relative to τ_i; the sum of lengths of all atomic sections in τ_j that access object θ is $\sum_{\theta \in (\theta_j \wedge \theta_i)} len(s_j(\theta))$; and $RC(T_{ji})$ is the $RC(T_j)$ without including the shared objects between τ_i and τ_j. The calculated WCET is relative to task τ_i, as it changes from task to task. The upper bound on the response time of τ_i, denoted R_i^{up}, can be calculated iteratively, using a modification of Theorem 6 in [8], as follows:

$$R_i^{up} = c_i + RC(T_i) + \left\lfloor \frac{1}{m} \sum_{j \neq i} W_{ij}(R_i^{up}) \right\rfloor \tag{3.11}$$

where R_i^{up}'s initial value is $c_i + RC(T_i)$.

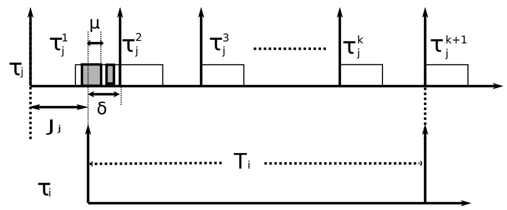


Figure 3.6: Atomic sections of job τ_j^1 contributing to period T_i

$W_{ij}(R_i^{up})$ is calculated by (3.3), and $W_{ij}(T_i)$ is calculated by (3.1), with c_j replaced by c_{ji}, and changing (3.2) as:

$$W_{ij}(L) = max \begin{cases} \left(\left\lceil \frac{L - \left(c_{ji} + \sum_{\theta \in (\theta_j \wedge \theta_i)} len(s_j(\theta)) \right)}{T_j} \right\rceil + 1 \right) c_{ji} \\ \left\lceil \frac{L - c_j}{T_j} \right\rceil .c_{ji} + c_j - \sum_{\theta \in (\theta_j \wedge \theta_i)} len(s_j(\theta)) \end{cases} \tag{3.12}$$

(3.12) compares two terms, as we have two cases:

Case 1. τ_j^1 (shown in Figure 3.2) contributes by c_{ji}. Thus, other instances of τ_j will begin after this modified WCET, but the sum of the shared objects' atomic section lengths is removed from c_{ji}, causing other instances to start earlier. Thus, the term $\sum_{\theta \in (\theta_i \wedge \theta_j)} len(s_j(\theta))$ is added to c_{ji} to obtain the correct start time.

Case 2. τ_j^1 contributes by its c_j, but the sum of the shared atomic section lengths between τ_i and τ_j should be subtracted from the contribution of τ_j^1, as they are already included in the retry cost.

It should be noted that subtraction of the sum of the shared objects' atomic section lengths is done in the first case to obtain the correct start time of other instances, while in the second case, this is done to get the correct contribution of τ_j^1. The maximum is chosen from the two terms in (3.12), because they differ in the contribution of their τ_j^1s, and the number of instances after that.

Tighter Upper Bound

To tighten τ_i's response time upper bound, $RC(\tau_i)$ needs to be calculated recursively over duration R_i^{up}, and not directly over T_i, as done in (3.11). So, (3.7) must be changed to include the modified number of interfering instances. And if R_i^{up} still extends to T_i, a situation like that shown in Figure 3.6 can happen.

To counter the situation in Figure 3.6, atomic sections of τ_j^1 that are contained in the interval δ are the only ones that can contribute to $RC(T_i)$. Of course, they can be lower, but cannot be greater, because τ_j^1 has been delayed by its maximum jitter. Hence, no more atomic sections can interfere during the duration $[d_j^1 - \delta, d_j^1]$.

For simplicity, we use the following notations:

- $\lambda_1(j,\theta) = \sum_{\forall s_j^l(\theta)\in\left[d_j^1-\delta,d_j^1\right]} len\left(s_j^{l^*}(\theta)+s_{max}(\theta)\right)$
- $\chi_1(i,j,\theta) = \left\lfloor \frac{T_i}{T_j} \right\rfloor \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta)+s_{max}(\theta)\right)$
- $\lambda_2(j,\theta) = \sum_{\forall s_j^l(\theta)\in\left[d_j^1-\delta,d_j^1\right]} len\left(s_j^{l^*}(\theta)+s_{max}^*(\theta)\right)$
- $\chi_2(i,j,\theta) = \left\lfloor \frac{T_i}{T_j} \right\rfloor \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta)+s_{max}^*(\theta)\right)$

Here, $s_j^{l^*}(\theta)$ is the part of $s_j^l(\theta)$ that is included in the interval δ. Thus, if $s_j^l(\theta)$ is partially included in δ, it contributes by its included length μ.

Now, (3.7) can be modified as:

$$RC(T_i) \leq \sum_{\theta\in\theta_i} min \begin{cases} \begin{cases} \left(\left(\sum_{\tau_j\in\gamma_i(\theta)} \lambda_1(j,\theta)+\chi_1(i,j,\theta)\right)\right. \\ \left. -s_{max}(\theta)+s_{i_{max}}(\theta)\right) \end{cases} \\ \begin{cases} \left(\left(\sum_{\tau_j\in\gamma_i(\theta)} \lambda_2(j,\theta)+\chi_2(i,j,\theta)\right)\right. \\ \left. -\bar{s}_{max}(\theta)+s_{i_{max}}(\theta)\right) \end{cases} \end{cases} \tag{3.13}$$

Now, we compute $RC(L)$, where L does not extend to the last instance of τ_j. Let:

- $\upsilon(L,j) = \left\lceil \frac{L-c_j}{T_j} \right\rceil + 1$
- $\lambda_3(j,\theta) = \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta)+s_{max}(\theta)\right)$
- $\lambda_4(j,\theta) = \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta)+s_{max}^*(\theta)\right)$

Now, (3.7) becomes:

$$RC(L) \leq \sum_{\theta\in\theta_i} min \begin{cases} \begin{cases} \left(\sum_{\tau_j\in\gamma_i(\theta)} \left(\upsilon(L,j)\,\lambda_3(j,\theta)\right)\right) \\ -s_{max}(\theta)+s_{i_{max}}(\theta) \end{cases} \\ \begin{cases} \left(\sum_{\tau_j\in\gamma_i(\theta)} \left(\upsilon(L,j)\,\lambda_4(j,\theta)\right)\right) \\ -\bar{s}_{max}(\theta)+s_{i_{max}}(\theta) \end{cases} \end{cases} \tag{3.14}$$

Thus, an upper bound on $RC(\tau_i)$ is given by:

$$RC(R_i^{up}) \leq min \begin{cases} RC(R_i^{up}) \\ RC(T_i) \end{cases} \tag{3.15}$$

where $RC(R_i^{up})$ is calculated by (3.14) if R_i^{up} does not extend to the last interfering instance of τ_j; otherwise, it is calculated by (3.13). The final upper bound on τ_i's response time can be calculated as in (3.11) by replacing $RC(T_i)$ with $RC(R_i^{up})$.

3.2 G-RMA/RMA CM Response Time

As G-RMA is a fixed priority scheduler, a task τ_i will be interfered by those tasks with priorities higher than τ_i (i.e., $p_j > p_i$). Upon a conflict, the RMA CM will commit the transaction that belongs to the higher priority task. Hereafter, we use *RCM* to refer to a multiprocessor system scheduled by G-RMA and resolves STM conflicts by the RMA CM. RCM is shown in Alogrithm 2.

Algorithm 2: RCM

Data: $s_i^k(\theta) \rightarrow$ interfered atomic section. $s_j^l(\theta) \rightarrow$ interfering atomic section
Result: which atomic section aborts
if $T_i < T_j$ **then**
  $s_j^l(\theta)$ aborts;
else
  $s_i^k(\theta)$ aborts;
end

3.2.1 Maximum Task Interference

Figure 3.7 illustrates the maximum interference caused by a task τ_j to a task τ_i under G-RMA. As τ_j is of higher priority than τ_i, τ_j^k will interfere with τ_i even if it is not totally included in T_i. Unlike the G-EDF case shown in Figure 3.6, where only the δ part of τ_j^1 is considered, in G-RMA, τ_j^k can contribute by the whole c_j, and all atomic sections contained in τ_j^k must be considered. This is because, in G-EDF, the worst-case pattern releases τ_i^a before d_j^1 by δ time units, and τ_i^a cannot be interfered before it is released. But in G-RMA, τ_i^a is already released, and can be interfered by the whole τ_j^k, even if this makes it infeasible.

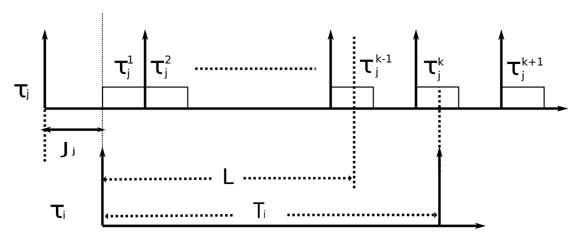


Figure 3.7: Max interference of τ_j to τ_i in G-RMA

Thus, the maximum contribution of τ_j^b to τ_i^a for any duration L can be deduced from Figure 3.7 as $W_{ij}(L) = \left(\left\lceil \frac{L-c_j}{T_j} \right\rceil + 1\right) c_j$, where L can extend to T_i. Note the contrast with ECM, where L cannot be extended directly to T_i, as this will have a different pattern of worst case interference from other tasks.

3.2.2 Retry Cost of Atomic Sections

Claim 3 *Under RCM, a task τ_i's retry cost over duration L, which can extend to T_i, is upper bounded by:*

$$\begin{aligned} RC\left(L\right) \;&\leq\; \sum_{\theta \in \theta_i} \left(\left(\sum_{\tau_j^*} \left(\left(\left\lceil \frac{L - c_j}{T_j} \right\rceil + 1 \right) \pi\left(j, \theta\right) \right) \right) \right. \\ &- \left. s_{max}^{min}\left(\theta\right) + s_{i_{max}}\left(\theta\right) \right) \end{aligned} \tag{3.16}$$

where:

- $\tau_j^* = \{\tau_j | (\tau_j \in \gamma_i(\theta)) \wedge (p_j > p_i)\}$
- $\pi(j,\theta) = \sum_{\forall s_j^l(\theta)} len\left(s_j^l\left(\theta\right) + s_{max}^j\left(\theta\right)\right)$
- $s_{max}^{min}(\theta) = min_{\forall \tau_j^*}\{s_{max}^j(\theta) \in \tau_k\}$, *where* $p_j > p_k > p_i$

Proof 3 *The worst case interference pattern for RCM is the same as that for ECM for an interval L, except that, in RCM, L can extend to the entire T_i, but in ECM, it cannot, as the interference pattern of τ_j to τ_i changes. Thus, (3.14) can be used to calculate τ_i's retry cost, with some modifications, as we do not have to obtain the minimum of the two terms in (3.14), because τ_j's atomic sections will abort and retry only atomic sections of tasks with lower priority than τ_j. Thus, $s_{max}(\theta)$, $s_{max}^*(\theta)$, and $\bar{s}_{max}(\theta)$ are replaced by $s_{max}^{min}(\theta)$, which is the minimum of the set of maximum-length atomic sections of tasks with priority lower than τ_j and share object θ with τ_i. This is because, the maximum length atomic section of tasks other than τ_j differs according to j. Besides, as τ_i's atomic sections can be aborted only by atomic sections of higher priority tasks, not all $\tau_j \in \gamma(\theta)$ are considered, but only the subset of tasks in $\gamma(\theta)$ with priority higher than τ_i (i.e., τ_j^*). Claim follows.*

3.2.3 Upper Bound on Response Time

The response time upper bound can be computed using Theorem 7 in [8] with a modification to include the effect of retry cost. The upper bound is given by:

$$R_i^{up} = c_i + RC(R_i^{up}) + \left\lfloor \frac{1}{m} \sum_{j \neq i} W_{ij}(R_i^{up}) \right\rfloor \tag{3.17}$$

where $W_{ij}(R_i^{up})$ is calculated as in (3.12), c_{ji} is calculated by (3.10), and RC is calculated by (3.16).

3.3 STM versus Lock-Free

We now would like to understand when STM will be beneficial compared to lock-free synchronization. The retry-loop lock-free approach in [25] is the most relevant to our work.

3.3.1 ECM versus Lock-Free

Claim 4 *For ECM's schedulability to be better or equal to that of [25]'s retry-loop lock-free approach, the size of s_{max} must not exceed one half of that of r_{max}, where r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task. With low number of conflicting tasks, the size of s_{max} can be at most the size of r_{max}.*

Proof 4 *Equation (3.15) can be upper bounded as:*

$$RC\left(T_i\right) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} \left(2.s_{max}\right) \right) \right) \tag{3.18}$$

where $s_j^l\left(\theta\right)$, $s_{i_{max}}\left(\theta\right)$, $s_{max}^*\left(\theta\right)$, and $\bar{s}_{max}\left(\theta\right)$ are replaced by s_{max}, and the order of the first two summations are reversed by each other, with γ_i being the set of tasks that share objects with τ_i. These changes are done to simplify the comparison.

Let $\sum_{\theta \in \theta_i} \sum_{\forall s_j^l(\theta)} = \beta_{i,j}^*$, and $\alpha_{edf} = \sum_{\tau_j \in \gamma_i} \left\lceil \frac{T_i}{T_j} \right\rceil .2\beta_{i,j}^*$. Now, (3.18) can be modified as:

$$RC\left(T_i\right) = \alpha_{edf}.s_{max} \tag{3.19}$$

The loop retry cost is given by:

$$\begin{aligned} LRC\left(T_i\right) &= \sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) .\beta_{i,j}.r_{max} \\ &= \alpha_{free}.r_{max} \end{aligned} \tag{3.20}$$

where $\beta_{i,j}$ is the number of retry loops of τ_j that accesses the same object as that accessed by some retry loop of τ_i, and $\alpha_{free} = \sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) .\beta_{i,j}$. Since the shared objects are the same in both STM and lock free, $\beta_{i,j} = \beta_{i,j}^*$. Thus, STM achieves equal or better schedulability than lock-free if the total utilization of the STM system is less than or equal to that of the lock-free system:

$$\begin{aligned} \sum_{\tau_i} \frac{c_i + \alpha_{edf}.s_{max}}{T_i} &\leq \sum_{\tau_i} \frac{c_i + \alpha_{free}.r_{max}}{T_i} \\ \therefore \frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i} \alpha_{free}/T_i}{\sum_{\tau_i} \alpha_{edf}/T_i} \end{aligned} \tag{3.21}$$

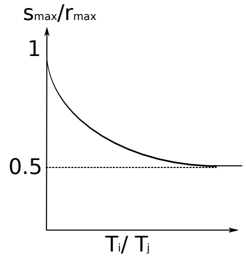


Figure 3.8: Effect of $\left\lceil \frac{T_i}{T_j} \right\rceil$ on $\frac{s_{max}}{r_{max}}$

Let $\bar{\alpha}_{free} = \sum_{\tau_j \in \gamma_i} \left\lceil \frac{T_i}{T_j} \right\rceil .\beta_{i,j}$, $\hat{\alpha}_{free} = \sum_{T_j \in \gamma_i} \beta_{i,j}$, *and* $\alpha_{free} = \bar{\alpha}_{free} + \hat{\alpha}_{free}$. *Therefore:*

$$\begin{aligned} \frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i} (\bar{\alpha}_{free} + \hat{\alpha}_{free})/T_i}{\sum_{\tau_i} \alpha_{edf}/T_i} \\ &= \frac{1}{2} + \frac{\sum_{\tau_i} \hat{\alpha}_{free}/T_i}{\sum_{\tau_i} \alpha_{edf}/T_i} \end{aligned} \tag{3.22}$$

Let $\zeta_1 = \sum_{\tau_i} \hat{\alpha}_{free}/T_i$ *and* $\zeta_2 = \sum_{\tau_i} \left(\frac{\alpha_{edf}}{2}\right)/T_i$. *The maximum value of* $\frac{\zeta_1}{2.\zeta_2} = \frac{1}{2}$, *which can happen if* $T_j \geq T_i \therefore \left\lceil \frac{T_i}{T_j} \right\rceil = 1$. *Then* (3.22) $= 1$, *which is its maximum value.* $T_j \geq T_i$ *means that there is a small number of interferences from other tasks to* τ_i, *and thus low number of conflicts. Therefore,* s_{max} *is allowed to be as large as* r_{max}.

The theoretical minimum value for $\frac{\zeta_1}{2.\zeta_2}$ *is* 0, *which can be asymptotically reached if* $T_j \ll T_i$, $\therefore \left\lceil \frac{T_i}{T_j} \right\rceil \rightarrow \infty$ *and* $\zeta_2 \rightarrow \infty$. *Thus,* (3.22) $\rightarrow 1/2$.

$\beta_{i,j}$ *has little effect on* s_{max}/r_{max}, *as it is contained in both the numerator and denominator. Irrespective of whether* $\beta_{i,j}$ *is going to reach its maximum or minimum value, both can be considered constants, and thus removed from (3.22)'s numerator and denominator. However, the number of interferences of other tasks to* τ_i, $\left\lceil \frac{T_i}{T_j} \right\rceil$, *has the main effect on* s_{max}/r_{max}. *This is illustrated in Figure 3.8. Claim follows.*

3.3.2 RCM versus Lock-Free

Claim 5 *For RCM's schedulability to be better or equal to that of [25]'s retry-loop lock-free approach, the size of* s_{max} *must not exceed one half of that of* r_{max} *for all cases. However, the size of* s_{max} *can be larger than that of* r_{max}, *depending on the number of accesses to a task* T_i*'s shared objects from other tasks.*

Proof 5 *Equation (3.16) is upper bounded by:*

$$\sum_{(\tau_j\in\gamma_i)\wedge(p_j>p_i)}\left(\left\lceil\frac{T_i-c_j}{T_j}\right\rceil+1\right).2.\beta_{i,j}.s_{max} \tag{3.23}$$

Consider the same assumptions as in Section 3.3.1. Let $\alpha_{rma}=\sum_{(\tau_j\in\gamma_i)\wedge(p_j>p_i)}\left(\left\lceil\frac{T_i-c_j}{T_j}\right\rceil+1\right).2.\beta_{i,j}$*. Now, the ratio* s_{max}/r_{max} *is upper bounded by:*

$$\frac{s_{max}}{r_{max}}\leq\frac{\sum_{T_i}\alpha_{free}/t\left(T_i\right)}{\sum_{T_i}\alpha_{rma}/t\left(T_i\right)} \tag{3.24}$$

The main difference between RCM and lock-free is that RCM is affected only by the higher priority tasks, while lock-free is affected by all tasks (just as in ECM). Besides, RCM is still affected by $2.\beta_{i,j}$ *(just as in ECM). The subtraction of* c_j *in the numerator of (3.23) may not have a significant effect on the ratio of (3.24), as the loop retry cost can also be modified to account for the effect of the first interfering instance of task* T_j*. Therefore,* $\alpha_{free}=\sum_{\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i-c_j}{T_j}\right\rceil+1\right)\beta_{i,j}$*.*

Let tasks in the denominator of (3.24) be given indexes k *instead of* i*, and* l *instead of* j*. Let tasks in both the numerator and denominator of (3.24) be arranged in the non-increasing priority order, so that* $i=k$ *and* $j=l$*. Let* α_{free} *in (3.24) be divided into two parts:* $\bar{\alpha}_{free}$ *that contains only tasks with priority higher than* τ_i*, and* $\hat{\alpha}_{free}$ *that contains only tasks with priority lower than* τ_i*. Now, (3.24) becomes:*

$$\begin{aligned}\frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i}(\bar{\alpha}_{free}+\hat{\alpha}_{free})/T_i}{\sum_{\tau_k}\alpha_{rma}/T_k}\\ &= \frac{1}{2}+\frac{\sum_{\tau_i}\hat{\alpha}_{free}/T_i}{\sum_{\tau_k}\alpha_{rma}/T_k}\end{aligned} \tag{3.25}$$

For convenience, we introduce the following notations:

$$\begin{aligned}\zeta_1 &= \sum_{\tau_i}\frac{\sum_{(\tau_j\in\gamma_i)\wedge(p_j<p_i)}\left(\left\lceil\frac{T_i-c_j}{T_j}\right\rceil+1\right)\beta_{i,j}}{T_i}\\ &= \sum_{T_i}\hat{\alpha}_{free}/T_i\\ \zeta_2 &= \sum_{\tau_k}\frac{\sum_{(\tau_l\in\gamma_k)\wedge(p_l>p_k)}\left(\left\lceil\frac{T_k-c_l}{T_l}\right\rceil+1\right)\beta_{k,l}}{T_k}\\ &= \frac{1}{2}\sum_{\tau_k}\alpha_{rma}/T_k\end{aligned}$$

τ_j is of lower priority than τ_i, which means $D_j > D_i$. Under G-RMA, this means, $T_j > T_i$. Thus, $\left\lceil \frac{T_i - c_j}{T_j} \right\rceil = 1$ for all τ_j and $\zeta_1 = \sum_{\tau_i}(\sum_{(\tau_j \in \gamma_i)\wedge(p_j<p_i)}(2.\beta_{i,j}))/T_i$. Since ζ_1 contains all τ_j of lower priority than τ_i and ζ_2 contains all τ_l of higher priority than τ_k, and tasks are arranged in the non-increasing priority order, then for each $\tau_{i,j}$, there exists $\tau_{k,l}$ such that $i = l$ and $j = k$. Figure 3.9 illustrates this, where 0 means that the pair i, j does not exist in ζ_1, and the pair k, l does not exist in ζ_2' (i.e., there is no task τ_l that will interfere with τ_k in ζ_2), and 1 means the opposite.

$$\begin{array}{cccccc} & j & 1 & 2 & \cdots & n \\ i & & & & & \\ 1 & & 0 & 1 & \cdots & 1 \\ 2 & & 0 & 0 & \ddots & \vdots \\ \vdots & & \vdots & \vdots & \ddots & 1 \\ n & & 0 & 0 & \cdots & 0 \end{array} \qquad \begin{array}{cccccc} & l & 1 & 2 & \cdots & n \\ k & & & & & \\ 1 & & 0 & 0 & \cdots & 0 \\ 2 & & 1 & 0 & & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ n & & 1 & \cdots & 1 & 0 \end{array}$$

Figure 3.9: Task association for lower priority tasks than T_i and higher priority tasks than T_k

Thus, it can be seen that both the matrices are transposes of each other. Consequently, for each $\beta_{i,j}$, there exists $\beta_{k,l}$ such that $i = l$ and $j = k$. But the number of times τ_j accesses a shared object with τ_i may not be the same as the number of times τ_i accesses that same object. Thus, $\beta_{i,j}$ does not have to be the same as $\beta_{k,l}$, even if i, j and k, l are transposes of each other. Therefore, we can analyze the behavior of s_{max}/r_{max} based on the three parameters $\beta_{i,j}$, $\beta_{k,l}$, and $\left\lceil \frac{T_k - c_l}{T_l} \right\rceil$. If $\beta_{i,j}$ is increased so that $\beta_{i,j} \to \infty$, $\therefore$ (3.25) $\to \infty$. This is because, $\beta_{i,j}$ represents the number of times a lower priority task τ_j accesses shared objects with a higher priority task τ_i. While this number has a greater effect in lock-free, it does not have any effect under RCM, because lower priority tasks do not affect higher priority ones. Hence, s_{max} is allowed to be much greater than r_{max}.

Although the minimum value for $\beta_{i,j}$ is 1, mathematically, if $\beta_{i,j} \to 0$, then (3.25) $\to 1/2$. Here, changing $\beta_{i,j}$ does not affect the retry cost of RCM, but it does affect the retry cost of lock-free, because the contention between tasks is reduced. Thus, s_{max} is reduced in this case to a little more than half of r_{max} ("a little more" because the minimum value of $\beta_{i,j}$ is actually 1, not 0).

The change of s_{max}/r_{max} with respect to $\beta_{i,j}$ is illustrated in Figure 3.10(a). If $\beta_{k,l} \to \infty$, then (3.25) $\to 1/2$. This is because, $\beta_{k,l}$ represents the number of times a higher priority task τ_l accesses shared objects with a lower priority task τ_k. Under RCM, this will increase the retry cost, thus reducing s_{max}/r_{max}. But if $\beta_{k,l} \to 0$, then (3.25)$\to \infty$. This is due to the lower contention from a higher priority task τ_l to a lower priority task τ_k, which reduces the retry cost under RCM and allows s_{max} to be very large compared with r_{max}. Of course, the actual minimum value for $\beta_{k,l}$ is 1, and is illustrated in Figure 3.10(b).

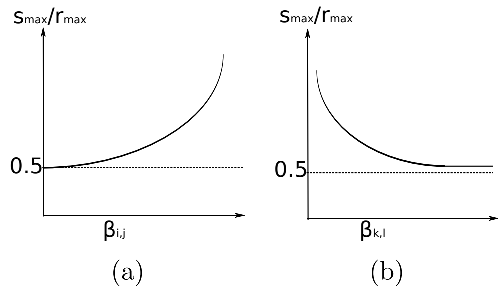


Figure 3.10: Change of s_{max}/r_{max}: a) $\frac{s_{max}}{r_{max}}$ versus $\beta_{i,j}$ and b) $\frac{s_{max}}{r_{max}}$ versus $\beta_{k,l}$

The third parameter that affects s_{max}/r_{max} is T_k/T_l. If $T_l \ll T_k$, then $\left\lceil \frac{T_k - c_l}{T_l} \right\rceil \to \infty$, and (3.25) $\to 1/2$. This is due to a high number of interferences from a higher priority task τ_l to a lower priority task τ_k, which increases the retry cost under RCM, and consequently reduces s_{max}/r_{max}.

If $T_l = T_k$ (which is the maximum value for T_l as $D_l \leq D_k$, because τ_l has a higher priority than τ_k), then $\left\lceil \frac{T_k - c_l}{T_l} \right\rceil \to 1$ and $\zeta_2 = \sum_{\tau_k} \frac{\sum_{(\tau_l \in \gamma_k) \wedge (p_l > p_k)} 2\beta_{k,l}}{t_k}$. This means that the system will be controlled by only two parameters, $\beta_{i,j}$ and $\beta_{k,l}$, as in the previous two cases, illustrated in Figures 3.10(a) and 3.10(b). Claim follows.

3.4 Conclusions

Under both ECM and RCM, a task incurs $2.s_{max}$ retry cost for each of its atomic sections due to a conflict with another task's atomic section. Retries under RCM and lock-free are affected by a larger number of conflicting task instances than under ECM. While task retries under ECM and lock-free are affected by all other tasks, retries under RCM are affected only by higher priority tasks.

STM and lock-free have similar parameters that affect their retry costs—i.e., the number of conflicting jobs and how many times they access shared objects. The s_{max}/r_{max} ratio determines whether STM is better or as good as lock-free. For ECM, this ratio cannot exceed 1, and it can be 1/2 for higher number of conflicting tasks. For RCM, for the common case, s_{max} must be 1/2 of r_{max}, and in some cases, s_{max} can be larger than r_{max} by many orders of magnitude.

Chapter 4

LCM

Under ECM and RCM, each atomic section can be aborted for at most $2.s_{max}$ by a single interfering atomic section. We present a novel contention manager (CM) for resolving transactional conflicts, called length-based CM (or LCM). LCM can reduce the abortion time of a single atomic section due to an interfering atomic section below $2.s_{max}$. We upper bound transactional retries and response times under LCM, when used with G-EDF and G-RMA schedulers. We identify the conditions under which LCM outperforms previous real-time STM CMs and lock-free synchronization. Our implementation and experimental studies reveal that G-EDF/LCM and G-RMA/LCM have shorter or comparable retry costs and response times than other synchronization techniques.

4.1 Length-based CM

LCM resolves conflicts based on the priority of conflicting jobs, besides the length of the interfering atomic section, and the length of the interfered atomic section. This is in contrast to ECM and RCM [31], where conflicts are resolved using the priority of the conflicting jobs. This strategy allows lower priority jobs, under LCM, to retry for lesser time than that under ECM and RCM, but higher priority jobs, sometimes, wait for lower priority ones with bounded priority-inversion.

4.1.1 Design and Rationale

For both ECM and RCM, $s_i^k(\theta)$ can be totally repeated if $s_j^l(\theta)$ — which belongs to a higher priority job τ_j^b than τ_i^a — conflicts with $s_i^k(\theta)$ at the end of its execution, while $s_i^k(\theta)$ is just about to commit. Thus, LCM, shown in Algorithm 3, uses the remaining length of $s_i^k(\theta)$ when it is interfered, as well as $len(s_j^l(\theta))$, to decide which transaction must be aborted.

Algorithm 3: LCM

Data: $s_i^k(\theta) \rightarrow$ interfered atomic section.
$s_j^l(\theta) \rightarrow$ interfering atomic section.
$\psi \rightarrow$ predefined threshold $\in [0, 1]$.
$\delta_i^k(\theta) \rightarrow$ remaining execution length of $s_i^k(\theta)$
Result: which atomic section of $s_i^k(\theta)$ or $s_j^l(\theta)$ aborts

1 **if** $p_i^k > p_j^l$ **then**
2     $s_j^l(\theta)$ aborts;
3 **else**
4     $c_{ij}^{kl} = len(s_j^l(\theta))/len(s_i^k(\theta))$;
5     $\alpha_{ij}^{kl} = ln(\psi)/(ln(\psi) - c_{ij}^{kl})$;
6     $\alpha = \left(len(s_i^k(\theta)) - \delta_i^k(\theta)\right)/len(s_i^k(\theta))$;
7   **if** $\alpha \leq \alpha_{ij}^{kl}$ **then**
8         $s_i^k(\theta)$ aborts;
9   **else**
10         $s_j^l(\theta)$ aborts;
11   **end**
12 **end**

If p_i^k was greater than p_j^l, then $s_i^k(\theta)$ would be the one that commits, because it belongs to a higher priority job, and it started before $s_j^l(\theta)$ (step 2). Otherwise, c_{ij}^{kl} is calculated (step 4) to determine whether it is worth aborting $s_i^k(\theta)$ in favor of $s_j^l(\theta)$, because $len(s_j^l(\theta))$ is relatively small compared to the remaining execution length of $s_i^k(\theta)$ (explained further).

We assume that:

$$c_{ij}^{kl} = len(s_j^l(\theta))/len(s_i^k(\theta)) \tag{4.1}$$

where $c_{ij}^{kl} \in]0, \infty[$, to cover all possible lengths of $s_j^l(\theta)$. Our idea is to reduce the opportunity for the abort of $s_i^k(\theta)$ if it is close to committing when interfered and $len(s_j^l(\theta))$ is large. This abort opportunity is increasingly reduced as $s_i^k(\theta)$ gets closer to the end of its execution, or $len(s_j^l(\theta))$ gets larger.

On the other hand, as $s_i^k(\theta)$ is interfered early, or $len(s_j^l(\theta))$ is small compared to $s_i^k(\theta)$'s remaining length, the abort opportunity is increased even if $s_i^k(\theta)$ is close to the end of its execution. To decide whether $s_i^k(\theta)$ must be aborted or not, we use a threshold value $\psi \in [0, 1]$ that determines α_{ij}^{kl} (step 5), where α_{ij}^{kl} is the maximum percentage of $len(s_i^k(\theta))$ below which $s_j^l(\theta)$ is allowed to abort $s_i^k(\theta)$. Thus, if the already executed part of $s_i^k(\theta)$ — when $s_j^l(\theta)$ interferes with $s_i^k(\theta)$ — does not exceed $\alpha_{ij}^{kl} len(s_i^k(\theta))$, then $s_i^k(\theta)$ is aborted (step 8). Otherwise, $s_j^l(\theta)$ is aborted (step 10).

The behavior of LCM is illustrated in Figure 4.1. In this figure, the horizontal axis corresponds to different values of α ranging from 0 to 1, and the vertical axis corresponds to different values of abort opportunities, $f(c_{ij}^{kl}, \alpha)$, ranging from 0 to 1 and calculated by (4.2):

$$f(c_{ij}^{kl}, \alpha) = e^{\frac{-c_{ij}^{kl}\alpha}{1-\alpha}} \tag{4.2}$$

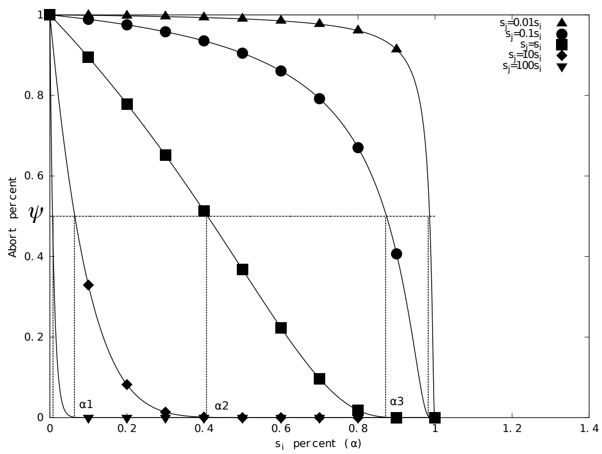


Figure 4.1: Interference of $s_i^k(\theta)$ by various lengths of $s_j^l(\theta)$

where c_{ij}^{kl} is calculated by (4.1).

Figure 4.1 shows one atomic section $s_i^k(\theta)$ (whose α changes along the horizontal axis) interfered by five different lengths of $s_j^l(\theta)$. For a predefined value of $f(c_{ij}^{kl}, \alpha)$ (denoted as ψ in Algorithm 3), there corresponds a specific value of α (which is α_{ij}^{kl} in Algorithm 3) for each curve. For example, when $len(s_j^l(\theta)) = 0.1 \times len(s_i^k(\theta))$, $s_j^l(\theta)$ aborts $s_i^k(\theta)$ if the latter has not executed more than $\alpha 3$ percentage (shown in Figure 4.1) of its execution length. As $len(s_j^l(\theta))$ decreases, the corresponding α_{ij}^{kl} increases (as shown in Figure 4.1, $\alpha 3 > \alpha 2 > \alpha 1$).

Equation (4.2) achieves the desired requirement that the abort opportunity is reduced as $s_i^k(\theta)$ gets closer to the end of its execution (as $\alpha \to 1$, $f(c_{ij}^{kl}, 1) \to 0$), or as the length of the conflicting transaction increases (as $c_{ij}^{kl} \to \infty$, $f(\infty, \alpha) \to 0$). Meanwhile, this abort opportunity is increased as $s_i^k(\theta)$ is interfered closer to its release (as $\alpha \to 0$, $f(c_{ij}^{kl}, 0) \to 1$), or as the length of the conflicting transaction decreases (as $c_{ij}^{kl} \to 0$, $f(0, \alpha) \to 1$).

LCM is not a centralized CM, which means that, upon a conflict, each transactions has to decide whether it must commit or abort.

Claim 6 *Let $s_j^l(\theta)$ interfere once with $s_i^k(\theta)$ at α_{ij}^{kl}. Then, the maximum contribution of $s_j^l(\theta)$ to $s_i^k(\theta)$'s retry cost is:*

$$W_i^k(s_j^l(\theta)) \leq \alpha_{ij}^{kl} len\Big(s_i^k(\theta)\Big) + len\Big(s_j^l(\theta)\Big) \tag{4.3}$$

Proof 6 *If $s_j^l(\theta)$ interferes with $s_i^k(\theta)$ at a Υ percentage, where $\Upsilon < \alpha_{ij}^{kl}$, then the retry cost of $s_i^k(\theta)$ is $\Upsilon len(s_i^k(\theta)) + len(s_j^l(\theta))$, which is lower than that calculated in (4.3). Besides, if $s_j^l(\theta)$ interferes with $s_i^k(\theta)$ after α_{ij}^{kl} percentage, then $s_i^k(\theta)$ will not abort.*

Claim 7 *An atomic section of a higher priority job, τ_j^b, may have to abort and retry due to a lower priority job, τ_i^a, if $s_j^l(\theta)$ interferes with $s_i^k(\theta)$ after the α_{ij}^{kl} percentage. τ_j's retry time, due to $s_i^k(\theta)$ and $s_j^l(\theta)$, is upper bounded by:*

$$W_j^l(s_i^k(\theta)) \leq \Big(1 - \alpha_{ij}^{kl}\Big) len\Big(s_i^k(\theta)\Big) \tag{4.4}$$

Proof 7 *It is derived directly from Claim 6, as $s_j^l(\theta)$ will have to retry for the remaining length of $s_i^k(\theta)$.*

Claim 8 *A higher priority job, τ_i^z, suffers from priority inversion for at most number of atomic sections in τ_i^z.*

Proof 8 *Assuming three atomic sections, $s_i^k(\theta)$, $s_j^l(\theta)$ and $s_a^b(\theta)$, where $p_j > p_i$ and $s_j^l(\theta)$ interferes with $s_i^k(\theta)$ after α_{ij}^{kl}. Then $s_j^l(\theta)$ will have to abort and retry. At this time, if $s_a^b(\theta)$ interferes with the other two atomic sections, and the LCM decides which transaction to commit based on comparison between each two transactions. So, we have the following cases:-*

- *$p_a < p_i < p_j$, then $s_a^b(\theta)$ will not abort any one because it is still in its beginning and it is of the lowest priority. So. τ_j is not indirectly blocked by τ_a.*
- *$p_i < p_a < p_j$ and even if $s_a^b(\theta)$ interferes with $s_i^k(\theta)$ before α_{ia}^{kb}, so, $s_a^b(\theta)$ is allowed abort $s_i^k(\theta)$. Comparison between $s_j^l(\theta)$ and $s_a^b(\theta)$ will result in LCM choosing $s_j^l(\theta)$ to commit and abort $s_a^b(\theta)$ because the latter is still beginning, and τ_j is of higher priority. If $s_a^b(\theta)$ is not allowed to abort $s_i^k(\theta)$, the situation is still the same, because $s_j^l(\theta)$ was already retrying until $s_i^k(\theta)$ finishes.*
- *$p_a > p_j > p_i$, then if $s_a^b(\theta)$ is chosen to commit, this is not priority inversion for τ_j because τ_a is of higher priority.*
- *if τ_a preempts τ_i, then LCM will compare only between $s_j^l(\theta)$ and $s_a^b(\theta)$. If $p_a < p_j$, then $s_j^l(\theta)$ will commit because of its task's higher priority and $s_a^b(\theta)$ is still at its beginning, otherwise, $s_j^l(\theta)$ will retry, but this will not be priority inversion because τ_a is already of higher priority than τ_j. If τ_a does not access any object but it preempts τ_i, then CM will choose $s_j^l(\theta)$ to commit as only already running transactions are competing together.*

So, by generalizing these cases to any number of conflicting jobs, it is seen that when an atomic section, $s_j^l(\theta)$, of a higher priority job is in conflict with a number of atomic sections belonging to lower priority jobs, $s_j^l(\theta)$ can suffer from priority inversion by only one of them. So, each higher priority job can suffer priority inversion at most its number of atomic section. Claim follows.

Claim 9 *The maximum delay suffered by $s_j^l(\theta)$ due to lower priority jobs is caused by the maximum length atomic section accessing object θ, which belongs to a lower priority job than τ_j^b that owns $s_j^l(\theta)$.*

Proof 9 *Assume three atomic sections, $s_i^k(\theta)$, $s_j^l(\theta)$, and $s_h^z(\theta)$, where $p_j > p_i$, $p_j > p_h$, and $len(s_i^k(\theta)) > len(s_h^z(\theta))$. Now, $\alpha_{ij}^{kl} > \alpha_{hj}^{zl}$ and $c_{ij}^{kl} < c_{hj}^{zl}$. By applying (4.4) to obtain the contribution of $s_i^k(\theta)$ and $s_h^z(\theta)$ to the priority inversion of $s_j^l(\theta)$ and dividing them, we get:*

$$\frac{W_j^l(s_i^k(\theta))}{W_j^l(s_h^z(\theta))} = \frac{\left(1-\alpha_{ij}^{kl}\right) len(s_i^k(\theta))}{\left(1-\alpha_{hj}^{zl}\right) len(s_h^z(\theta))}$$

By substitution for αs from (4.2):

$$= \frac{(1-\frac{ln\psi}{ln\psi - c_{ij}^{kl}})len(s_i^k(\theta))}{(1-\frac{ln\psi}{ln\psi - c_{hj}^{zl}})len(s_h^z(\theta))} = \frac{(\frac{-c_{ij}^{kl}}{ln\psi - c_{ij}^{kl}})len(s_i^k(\theta))}{(\frac{-c_{hj}^{zl}}{ln\psi - c_{hj}^{zl}})len(s_h^z(\theta))}$$

$\because ln\psi \leq 0$ and $c_{ij}^{kl}, c_{hj}^{kl} > 0, \therefore$ by substitution from (4.1)

$$= \frac{len(s_j^l(\theta))/(ln\psi - c_{ij}^{kl})}{len(s_j^l(\theta))/(ln\psi - c_{hj}^{zl})} = \frac{ln\psi - c_{hj}^{zl}}{ln\psi - c_{ij}^{kl}} > 1$$

Thus, as the length of the interfered atomic section increases, the delay suffered by the interfering atomic section increases. Claim follows.

4.1.2 Response Time of G-EDF/LCM

Claim 10 *$RC(T_i)$ for a task τ_i under G-EDF/LCM is upper bounded by:*

$$\begin{aligned} RC(T_i) &= \Bigg(\sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in \theta_i \wedge \theta_h} \Bigg(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^l(\theta)} len\Big(s_h^l(\theta)\Big) \\ &+ \alpha_{max}^{hl} len\Big(s_{max}^h(\theta)\Big)\Bigg)\Bigg) \\ &+ \sum_{\forall s_i^y(\theta)} \Big(1-\alpha_{max}^{iy}\Big) len\Big(s_{max}^i(\theta)\Big) \end{aligned} \tag{4.5}$$

where α_{max}^{hl} is the α value that corresponds to ψ due to the interference of $s_{max}^h(\theta)$ by $s_h^l(\theta)$. α_{max}^{iy} is the α value that corresponds to ψ due to the interference of $s_{max}^i(\theta)$ by $s_i^y(\theta)$.

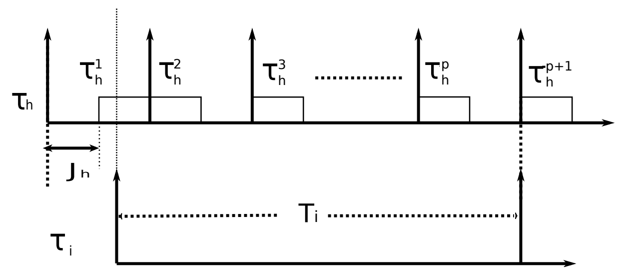


Figure 4.2: τ_h^p has a higher priority than τ_i^x

Proof 10 *The maximum number of higher priority instances of* τ_h *that can interfere with* τ_i^x *is* $\left\lceil \frac{T_i}{T_h} \right\rceil$, *as shown in Figure 4.2, where one instance of* τ_h *and* τ_h^p *coincides with the absolute deadline of* τ_i^x.

By using Claims 6, 7, 8, and 9, and Claim 1 in [31] to determine the effect of atomic sections belonging to higher and lower priority instances of interfering tasks to τ_i^x, *Claim follows.*

Response time of τ_i is calculated by (11) in [31].

4.1.3 Schedulability of G-EDF/LCM and ECM

We now compare the schedulability of G-EDF/LCM with ECM [31] to understand when G-EDF/LCM will perform better. Toward this, we compare the total utilization of ECM with that of G-EDF/LCM. For each method, we inflate the c_i of each task τ_i by adding the retry cost suffered by τ_i. Thus, if method A adds retry cost $RC_A(T_i)$ to c_i, and method B adds retry cost $RC_B(T_i)$ to c_i, then the schedulability of A and B are compared as:

$$\begin{aligned}\sum_{\forall \tau_i} \frac{c_i + RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{c_i + RC_B(T_i)}{T_i} \\ \sum_{\forall \tau_i} \frac{RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{RC_B(T_i)}{T_i} \end{aligned} \tag{4.6}$$

Thus, schedulability is compared by substituting the retry cost added by the synchronization methods in (4.6).

Claim 11 *Let* s_{max} *be the maximum length atomic section accessing any object* θ. *Let* α_{max} *and* α_{min} *be the maximum and minimum values of* α *for any two atomic sections* $s_i^k(\theta)$ *and* $s_j^l(\theta)$. *Given a threshold* ψ, *schedulability of G-EDF/LCM is equal or better than ECM if for any task* τ_i:

$$\frac{1-\alpha_{min}}{1-\alpha_{max}} \leq \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil \tag{4.7}$$

Proof 11 *Under ECM, $RC(T_i)$ is upper bounded by:*

$$RC(T_i) \leq \sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in \ (\theta_i \wedge \theta_h)} \left(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^z(\theta)} 2len(s_{max}) \right) \tag{4.8}$$

with the assumption that all lengths of atomic sections of (4) and (8) in [31] and (4.5) are replaced by s_{max}. Let α_{max}^{hl} in (4.5) be replaced with α_{max}, and α_{max}^{iy} in (4.5) be replaced with α_{min}. As α_{max}, α_{min}, and $len(s_{max})$ are all constants, (4.5) is upper bounded by:

$$\begin{aligned} RC(T_i) \ \leq \ & \left(\sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in \theta_i \wedge \theta_h} \left(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^l(\theta)} (1+\alpha_{max}) \right.\right. \\ & \left.\left. len\Big(s_{max}\Big)\right)\right) + \sum_{\forall s_i^y(\theta)} \Big(1-\alpha_{min}\Big) len\Big(s_{max}\Big) \end{aligned} \tag{4.9}$$

If β_1^{ih} is the total number of times any instance of τ_h accesses shared objects with τ_i, then $\beta_1^{ih} = \sum_{\forall \theta \in (\theta_i \wedge \theta_h)} \sum_{\forall s_h^z(\theta)}$. Furthermore, if β_2^i is the total number of times any instance of τ_i accesses shared objects with any other instance, $\beta_2^i = \sum_{\forall s_i^y(\theta)}$, where θ is shared with another task. Then, $\beta_i = max\{max_{\forall \tau_h \in \gamma_i}\{\beta_1^{ih}\}, \beta_2^i\}$ is the maximum number of accesses to all shared objects by any instance of τ_i or τ_h. Thus, (4.8) becomes:

$$RC(T_i) \leq \sum_{\tau_h \in \gamma_i} 2 \left\lceil \frac{T_i}{T_h} \right\rceil \beta_i len(s_{max}) \tag{4.10}$$

and (4.9) becomes:

$$\begin{aligned} RC(T_i) \ \leq \ & \beta_i len(s_{max}) \Bigg((1-\alpha_{min}) \\ & + \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil (1+\alpha_{max}) \Bigg) \end{aligned} \tag{4.11}$$

We can now compare the total utilization of G-EDF/LCM with that of ECM by comparing (4.9) and (4.11) for all τ_i:

$$\begin{aligned} & \sum_{\forall \tau_i} \frac{(1-\alpha_{min}) + \sum_{\forall \tau_h \in \gamma_i} \left(\left\lceil \frac{T_i}{T_h} \right\rceil (1+\alpha_{max}) \right)}{T_i} \\ \leq \ & \sum_{\forall \tau_i} \frac{\sum_{\forall \tau_h \in \gamma_i} 2 \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i} \end{aligned} \tag{4.12}$$

(4.12) is satisfied if for each τ_i, the following condition is satisfied:

$$(1-\alpha_{min})+\sum_{\forall\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil(1+\alpha_{max})\right)\leq 2\sum_{\forall\tau_h\in\gamma_i}\left\lceil\frac{T_i}{T_h}\right\rceil$$

$$\therefore\frac{1-\alpha_{min}}{1-\alpha_{max}}\leq\sum_{\forall\tau_h\in\gamma_i}\left\lceil\frac{T_i}{T_h}\right\rceil$$

Claim follows.

4.1.4 G-EDF/LCM versus Lock-free

We consider the retry-loop lock-free synchronization for G-EDF given in [25]. This lock-free approach is the most relevant to our work.

Claim 12 *Let s_{max} denote $len(s_{max})$ and r_{max} denote the maximum execution cost of a single iteration of any retry loop of any task in the retry-loop lock-free algorithm in [25]. Now, G-EDF/LCM achieves higher schedulability than the retry-loop lock-free approach if the upper bound on s_{max}/r_{max} under G-EDF/LCM ranges between 0.5 and 2 (which is higher than that under ECM).*

Proof 12 *From [25], the retry-loop lock-free algorithm is upper bounded by:*

$$RL(T_i)=\sum_{\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil+1\right)\beta_i r_{max}\tag{4.13}$$

where β_i is as defined in Claim 11. The retry cost of τ_i in G-EDF/LCM is upper bounded by (4.11). By comparing G-EDF/LCM's total utilization with that of the retry-loop lock-free algorithm, we get:

$$\sum_{\forall\tau_i}\frac{\left((1-\alpha_{min})+\sum_{\forall\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil(1+\alpha_{max})\right)\right)\beta_i s_{max}}{T_i}$$
$$\leq\quad\sum_{\forall\tau_i}\frac{\sum_{\forall\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil+1\right)\beta_i r_{max}}{T_i}$$

$$\therefore\frac{s_{max}}{r_{max}}\leq\frac{\sum_{\forall\tau_i}\frac{\sum_{\forall\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil+1\right)\beta_i}{T_i}}{\sum_{\forall\tau_i}\frac{\left((1-\alpha_{min})+\sum_{\forall\tau_h\in\gamma_i}\left(\left\lceil\frac{T_i}{T_h}\right\rceil(1+\alpha_{max})\right)\right)\beta_i}{T_i}}\tag{4.14}$$

Let the number of tasks that have shared objects with τ_i be ω (i.e., $\sum_{\tau_h\in\gamma_i}=\omega\geq 1$ since at least one task has a shared object with τ_i; otherwise, there is no conflict between tasks). Let *the total number of tasks be n, so* $1 \leq \omega \leq n-1$, *and* $\left\lceil \frac{T_i}{T_h} \right\rceil \in [1, \infty[$. *To find the minimum and maximum values for the upper bound on* s_{max}/r_{max}, *we consider the following cases:*

- $\alpha_{min} \rightarrow 0, \alpha_{max} \rightarrow 0$

$\therefore$ *(4.14) will be:*

$$\frac{s_{max}}{r_{max}} \leq 1 + \frac{\sum_{\forall \tau_i} \frac{\omega - 1}{T_i}}{\sum_{\forall \tau_i} \frac{1 + \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{4.15}$$

By substituting the edge values for ω *and* $\left\lceil \frac{T_i}{T_h} \right\rceil$ *in (4.15), we derive that the upper bound on* s_{max}/r_{max} *lies between 1 and 2.*

- $\alpha_{min} \rightarrow 0, \alpha_{max} \rightarrow 1$

(4.14) becomes

$$\frac{s_{max}}{r_{max}} \leq 0.5 + \frac{\sum_{\forall \tau_i} \frac{\omega - 0.5}{T_i}}{\sum_{\forall \tau_i} \frac{1 + 2\sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{4.16}$$

By applying the edge values for ω *and* $\left\lceil \frac{T_i}{T_h} \right\rceil$ *in (4.16), we derive that the upper bound on* s_{max}/r_{max} *lies between 0.5 and 1.*

- $\alpha_{min} \rightarrow 1, \alpha_{max} \rightarrow 0$

This case is rejected since $\alpha_{min} \leq \alpha_{max}$.

- $\alpha_{min} \rightarrow 1, \alpha_{max} \rightarrow 1$

$\therefore$ *(4.14) becomes:*

$$\frac{s_{max}}{r_{max}} \leq 0.5 + \frac{\sum_{\tau_i} \frac{\omega}{T_i}}{2\sum_{\tau_i} \frac{\sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{4.17}$$

By applying the edge values for ω *and* $\left\lceil \frac{T_i}{T_h} \right\rceil$ *in (4.17), we derive that the upper bound on* s_{max}/r_{max} *lies between 0.5 and 1, which is similar to that achieved by ECM.*

Summarizing from the previous cases, the upper bound on s_{max}/r_{max} *lies between 0.5 and 2, whereas for ECM [31], it lies between 0.5 and 1. Claim follows.*

4.1.5 Response Time of G-RMA/LCM

Claim 13 *Let* $\lambda_2(j,\theta) = \sum_{\forall s_j^l(\theta)} len(s_j^l(\theta)) + \alpha_{max}^{jl} len(s_{max}^j(\theta))$*, where* α_{max}^{jl} *is the* α *value corresponding to* ψ *due to the interference of* $s_{max}^j(\theta)$ *by* $s_j^l(\theta)$*. The retry cost of any task* τ_i *under G-RMA/LCM during* T_i *is given by:*

$$\begin{aligned} RC\left(T_i\right) &= \sum_{\forall \tau_j^*} \left(\sum_{\theta \in (\theta_i \wedge \theta_j)} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \lambda_2(j,\theta) \right) \right) \\ &+ \sum_{\forall s_i^y(\theta)} \left(1 - \alpha_{max}^{iy}\right) len\left(s_{max}^i(\theta)\right) \end{aligned} \tag{4.18}$$

where $\tau_j^* = \{\tau_j | (\tau_j \in \gamma_i) \wedge (p_j > p_i)\}$.

Proof 13 *Under G-RMA, all instances of a higher priority task,* τ_j*, can conflict with a lower priority task,* τ_i*, during* T_i*. (4.3) can be used to determine the contribution of each conflicting atomic section in* τ_j *to* τ_i*. Meanwhile, all instances of any task with lower priority than* τ_i *can conflict with* τ_i *during* T_i*. Claims 7 and 8 can be used to determine the contribution of conflicting atomic sections in lower priority tasks to* τ_i*. Using the previous notations and Claim 3 in [31], the Claim follows.*

The response time is calculated by (17) in [31] with replacing $RC(R_i^{up})$ with $RC(T_i)$.

4.1.6 Schedulability of G-RMA/LCM and RCM

Claim 14 *Under the same assumptions of Claims 11 and 13, G-RMA/LCM's schedulability is equal or better than RCM if:*

$$\frac{1-\alpha_{min}}{1-\alpha_{max}} \leq \sum_{\forall \tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \tag{4.19}$$

Proof 14 *Under the same assumptions as that of Claims 11 and 13, (4.18) can be upper bounded as:*

$$\begin{aligned} RC(T_i) &\leq \sum_{\forall \tau_j^*} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) (1+\alpha_{max}) len(s_{max}) \beta_i \right) \\ &+ (1-\alpha_{min}) len(s_{max}) \beta_i \end{aligned} \tag{4.20}$$

For RCM, (16) in [31] for $RC(T_i)$ *is upper bounded by:*

$$RC(T_i) \leq \sum_{\forall \tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) 2\beta_i len(s_{max})$$

By comparing the total utilization of G-RMA/LCM with that of RCM, we get:

$$\begin{aligned}&\sum_{\forall\tau_i}\frac{len(s_{max})\beta_i\left((1-\alpha_{min})+\sum_{\forall\tau_j^*}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)(1+\alpha_{max})\right)\right)}{T_i}\\ \leq\quad&\sum_{\forall\tau_i}\frac{2len(s_{max})\beta_i\sum_{\forall\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)}{T_i}\end{aligned}\tag{4.21}$$

(4.21) is satisfied if $\forall\tau_i$ *(4.19) is satisfied. Claim follows.*

4.1.7 G-RMA/LCM versus Lock-free

Although [25] considers retry-loop lock-free synchronization for G-EDF systems, [25] also applies for G-RMA systems.

Claim 15 *Let* s_{max} *denote* $len(s_{max})$ *and* r_{max} *denote the maximum execution cost of a single iteration of any retry loop of any task in the retry-loop lock-free algorithm in [25]. G-RMA/LCM achieves higher schedulability than the retry-loop lock-free approach if the upper bound on* s_{max}/r_{max} *under G-RMA/LCM is no less than 0.5. Upper bound on* s_{max}/r_{max} *can extend to large values when* α_{min} *and* α_{max} *are very large.*

Proof 15 *The retry cost for G-RMA/LCM is upper bounded by (4.18). Let* $\gamma_i = \tau_j^* \cup \bar{\tau}_j$, *where* τ_j^* *is the set of higher priority tasks than* τ_i *sharing objects with* τ_i. $\bar{\tau}_j$ *is the set of lower priority tasks than* τ_i *sharing objects with it. We follow the same definitions of* β_i, r_{max}, *and* $RL(T_i)$ *given in the proof of Claim (12). Schedulability of G-RMA/LCM equals or exceeds the schedulability of retry-loop lock-free algorithm if:*

$$\begin{aligned}\frac{s_{max}}{r_{max}}\leq\;&\frac{\sum_{\forall\tau_i}\frac{\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)}{T_i}}{\sum_{\forall\tau_i}\frac{\left(1-\alpha_{min}\right)+\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)(1+\alpha_{max})}{T_i}}\\ +\;&\frac{2\sum_{\forall\tau_i}\frac{\sum_{\forall\bar{\tau}_j}}{T_i}}{\sum_{\forall\tau_i}\frac{\left(1-\alpha_{min}\right)+\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)(1+\alpha_{max})}{T_i}}\end{aligned}\tag{4.22}$$

If $p_j < p_i$, $\therefore \left\lceil\frac{T_i}{T_j}\right\rceil = 1$, *because the system assumes implicit deadline tasks and uses the G-RMA scheduler. Let* ω_1 *be the size of* τ_i^* *and* ω_2 *be the size of* $\bar{\tau}_i$. $\therefore \omega_1^i \geq 1$ *and* $\omega_2^i \geq 1$. *Otherwise, there is no conflict with* τ_i. *To find the maximum and minimum upper bounds for* s_{max}/r_{max}, *the following cases are considered:*

- $\alpha_{min} \to 0$, $\alpha_{max} \to 0$

$$\therefore \frac{s_{max}}{r_{max}} \leq 1 + \frac{\sum_{\forall \tau_i} \frac{2\omega_2^i - 1}{T_i}}{\sum_{\forall \tau_i} \frac{1 + \sum_{\tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)}{T_i}} \tag{4.23}$$

As the second term in (4.23) is always positive (because $\omega_2^i \geq 1$), the minimum upper bound on s_{max}/r_{max} is 1. To get the maximum upper bound on s_{max}/r_{max}, let $\left\lceil \frac{T_i}{T_j} \right\rceil$ approach its minimum value of 1, $\omega_1^i \to 0$, and $\omega_2^i \to n-1$ (the maximum and minimum values for ω_1^i and ω_2^i, respectively. n is number of tasks). Now:

$$\therefore \frac{s_{max}}{r_{max}} \leq (2n - 2)$$

Of course, n cannot be lower than 2. Otherwise, there will be no conflicting tasks.

- $\alpha_{min} \to 0$, $\alpha_{max} \to 1$

$$\frac{s_{max}}{r_{max}} \leq \frac{1}{2} + \frac{\sum_{\forall \tau_i} \frac{4\omega_2^i - 1}{T_i}}{2 \sum_{\forall \tau_i} \frac{1 + 2 \sum_{\tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)}{T_i}} \tag{4.24}$$

The minimum upper bound for s_{max}/r_{max} is 0.5. This can happen when $T_i \gg T_j$. To get the maximum upper bound on s_{max}/r_{max}, let $\left\lceil \frac{T_i}{T_j} \right\rceil$ approach its minimum value 1, $\omega_2^i \to n - 1$, and $\omega_1^i \to 0$. Now:

$$\frac{s_{max}}{r_{max}} \leq 2n - 2$$

- $\alpha_{min} \to 1$, $\alpha_{max} \to 0$ *This case is rejected because α_{max} must be greater or equal to α_{min}.*

- $\alpha_{min} \to 1$, $\alpha_{max} \to 1$

$$\frac{s_{max}}{r_{max}} \leq \frac{1}{2} + \frac{\sum_{\forall \tau_i} \frac{\omega_2^i}{T_i}}{\sum_{\forall \tau_i} \frac{\sum_{\tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)}{T_i}} \tag{4.25}$$

The minimum upper bound for s_{max}/r_{max} is 0.5. This can happen when $T_i \gg T_j$. To get the maximum upper bound on s_{max}/r_{max}, let $\left\lceil \frac{T_i}{T_j} \right\rceil$ approach its minimum value 1, $\omega_2^i \to n - 1$, $\omega_1^i \to 0$. Now:

$$\frac{s_{max}}{r_{max}} \to \infty$$

From the previous cases, we can derive that the upper bound on s_{max}/r_{max} extends from 0.5 to large values. Claim follows.

4.2 Conclusions

In ECM and RCM, a task incurs at most $2s_{max}$ retry cost for each of its atomic section due to conflict with another task's atomic section. With LCM, this retry cost is reduced to $(1+\alpha_{max})s_{max}$ for each aborted atomic section. In ECM and RCM, tasks do not retry due to lower priority tasks, whereas in LCM, they do so. In G-EDF/LCM, retry due to a lower priority job is encountered only from a task τ_j's last job instance during τ_i's period. This is not the case with G-RMA/LCM, because, each higher priority task can be aborted and retried by any job instance of lower priority tasks. Schedulability of G-EDF/LCM and G-RMA/LCM is better or equal to ECM and RCM, respectively, by proper choices for α_{min} and α_{max}. Schedulability of G-EDF/LCM is better than retry-loop lock-free synchronization for G-EDF if the upper bound on s_{max}/r_{max} is between 0.5 and 2, which is higher than that achieved by ECM. G-RMA/LCM achieves higher schedulability than retry-loop lock-free synchronization if s_{max}/r_{max} is not less than 0.5. For high values of α in G-RMA/LCM, s_{max}/r_{max} can extend to large values.

Chapter 5

PNF

In this chapter, we present a novel contention manager for resolving transactional conflicts, called PNF. We upper bound transactional retries and task response times under PNF, when used with the G-EDF and G-RMA schedulers. We formally identify the conditions under which PNF outperforms previous real-time STM contention managers and lock-free synchronization. We also implement PNF and competitor techniques in the Rochester STM framework and conduct experimental studies using a real-time Linux kernel to understand average-case performance. Our work reveals that G-EDF/PNF and G-RMA/PNF have shorter or comparable retry costs than other synchronization techniques.

5.1 ECM, RCM and LCM: Limitations

ECM and RCM [31] use dynamic and fixed priorities, respectively, to resolve transactional conflicts. ECM is used with the G-EDF scheduler, and allows the transaction whose job has the earliest absolute deadline to commit first [32]. RCM is used with the G-RMA scheduler, and allows the transaction whose job has the shortest relative deadline to commit first.

G-EDF/LCM [30] and G-RMA/LCM act as ECM and RCM respectively with some difference. Under LCM, higher priority transaction $s_i^k(\theta)$ cannot abort a lower priority transaction $s_j^l(\theta)$ if $s_j^l(\theta)$ has already passed α percentage of its length.

As mentioned before, [30, 31] assumes that each transaction accesses only one object. This assumption simplifies the retry cost (Claims 2 and 3 in [31], and Claims 5, 8 in [30]) and response time analysis (Sections 4 and 5 in [31], and Sections 4.2, 4.5 in [30]). Besides, it enables a one-to-one comparison with lock-free synchronization in [25]. With multiple objects per transaction, [31] and [30] will introduce transitive retry, which we illustrate with an example.

Example 1. Consider three atomic sections s_1^x, s_2^y, and s_3^z belonging to jobs τ_1^x,τ_2^y, and

τ_3^z, with priorities $p_3^z > p_2^y > p_1^x$, respectively. Assume that s_1^x and s_2^y share objects, s_2^y and s_3^z share objects. s_1^x and s_3^z do not share objects. s_3^z can cause s_2^y to retry, which in turn will cause s_1^x to retry. This means that s_1^x may retry transitively because of s_3^z, which will increase the retry cost of s_1^x.

Assume another atomic section s_4^f is introduced. Priority of s_4^f is higher than priority of s_3^z. s_4^f shares objects only with s_3^z. Thus, s_4^f can make s_3^z to retry, which in turn will make s_2^y to retry, and finally, s_1^x to retry. Thus, transitive retry will move from s_4^f to s_1^x, increasing the retry cost of s_1^x. The situation gets worse as more tasks of higher priorities are added, where each task shares objects with its immediate lower priority task. τ_3^z may have atomic sections that share objects with τ_1^x, but this will not prevent the effect of transitive retry due to s_1^x.

Definition 1 ***Transitive Retry:*** *A transaction s_i^k suffers from transitive retry when it conflicts with a higher priority transaction s_j^l, which in turn conflicts with a higher priority transaction s_z^h, but s_i^k does not conflict with s_z^h. Still, when s_j^l retries due to s_z^h, s_i^k also retries due to s_j^l. Thus, the effect of the higher priority transaction s_z^h is transitively moved to the lower priority transaction s_i^k, even when they do not conflict on common objects.*

Claim 16 *ECM, RCM and LCM suffer from transitive retry for multi-object transactions.*

Proof 16 *ECM, RCM and LCM depend on priorities to resolve conflicts between transactions. Thus, lower priority transaction must always be aborted for a conflicting higher priority transaction in ECM and RCM. In LCM, lower priority transactions are conditionally aborted for higher priority ones. Claim follows.*

Therefore, the analysis in [31] and [30] must extend the set of objects that can cause an atomic section of a lower priority job to retry. This can be done by initializing the set of conflicting objects, γ_i, to all objects accessed by all transactions of τ_i. We then cycle through all transactions belonging to all other higher priority tasks. Each transaction s_j^l that accesses at least one of the objects in γ_i adds all other objects accessed by s_j^l to γ_i. The loop over all higher priority tasks is repeated, each time with the new γ_i, until there are no more transactions accessing any object in γ_i[1].

In addition to the *transitive retry* problem, retrying higher priority transactions can prevent lower priority tasks from running. This happens when all processors are busy with higher priority jobs. When a transaction retries, the processor time is wasted. Thus, it would be better to give the processor to some other task.

Essentially, what we present is a new contention manager that avoids the effect of transitive retry. We call it, Priority contention manager with Negative values and First access (or PNF). PNF also tries to enhance processor utilization. This is done by allocating processors to jobs with non-retrying transactions. PNF is described in detail in Section 5.2.

[1]However, note that, this solution may over-extend the set of conflicting objects, and may even contain all objects accessed by all tasks.

5.2 The PNF Contention Manager

Algorithm 4: PNF Algorithm

Data: *Executing Transaction:* is one that cannot be aborted by any other transaction, nor preempted by a higher priority task;
m-set: m-length set that contains only non-conflicting executing transactions;
n-set: n-length set that contains retrying transactions for n tasks in non-increasing order of priority;
n(z): transaction at index z of the n-set;
s_i^k: a newly released transaction;
s_j^l: one of the executing transactions;
Result: atomic sections that will commit

1 **if** s_i^k *does not conflict with any executing transaction* **then**
2   Assign s_i^k as an executing transaction;
3   Add s_i^k to the m-set;
4   Select s_i^k to commit
5 **else**
6   Add s_i^k to the n-set according to its priority;
7   Assign temporary priority -1 to the job that owns s_i^k ;
8   Select transaction(s) conflicting with s_i^k for commit;
9 **end**
10 **if** s_j^l *commits* **then**
11   **for** *z=1 to size of n-set* **do**
12   **if** *n(z) does not conflict with any executing transaction* **then**
13   **if** *processor available* **then**
14   Restore priority of task owning n(z);
15   Assign n(z) as executing transaction;
16   Add n(z) to m-set;
17   Select n(z) for commit;
18   **end**
19   **end**
20   move to the next n(z);
21   **end**
22 **end**

Algorithm 4 describes PNF. It manages two sets. The first is the m-set, which contains at most m non-conflicting transactions, where m is the number of processors, as there cannot be more than m executing transactions (or generally, m executing jobs) at the same time. When a transaction is entered in the m-set, it executes non-preemptively and no other transaction can abort it. A transaction in the m-set is called an *executing transaction*. This means that, when a transaction is executing before the arrival of higher priority conflicting transactions, then the one that started executing first will be committed (Step 8) (hence the word "First" in the algorithm's name).

The second set is the n-set, which holds the transactions that are retrying because of a conflict with one or more of the executing transactions (Step 6), where n stands for the number of tasks in the system. It also holds transactions that cannot currently execute, because processors are busy, either due to processing preempted transaction and/or higher priority jobs. Any transaction in the n-set is assigned a temporal priority of -1 (Step 7) (hence the word “Negative” in the algorithm’s name). A negative priority is considered smaller than any normal priority, and a transaction continues to hold this negative priority until it is moved to the m-set, where it is restored its normal priority.

A job holding a transaction in the n-set can be preempted by any other job with normal priority, even if that normal priority job does not have transactions conflicting with the preempted job. Hence, this set is of length n, as there can be at most n jobs in the system at the same time. Transactions in the n-set whose jobs have been preempted are called preempted transactions. The n-set list keeps track of preempted transactions, because as it will be shown, preempted transactions are examined when any of the executing transaction commits. Then, one or more transactions are selected from the m-set to be preempted transaction. If a preempted transaction is selected as an executing transaction, then the task that owns the preempted transaction regains its priority. Thus, an aborted transaction can preempt the job which previously preempted it when the transaction was in the n-set.

When a new transaction is released, and it does not conflict with any of the preempted transaction (Step 1), then it will allocate a slot in the m-set and becomes an executing transaction itself. When this transaction is released (which means that its containing task is already allocated to a processor), it will be able to access a processor immediately. This new transaction may have a conflict with any of the transactions in the n-set. However, since transactions in the n-set have priorities of -1, they cannot prevent this new transaction from executing if it does not conflict with any of the preempted transaction.

When one of the preempted transaction commits (Step 10), it is time to select one of the n-set transactions to commit. The n-set is traversed from the highest priority transaction to the lowest priority (where priority here refers to the original priority of the transactions, and not -1) (Step 11).

If an examined transaction in the n-set, s_h^b, does not conflict with any executing transaction (Step 12), and there is an available processor for it (Step 13) (where “available” means either an idle processor, or one that is executing a job of lower priority than s_h^b), then s_h^b is moved from the n-set to the m-set, as an executing transaction and restored its original priority.

If s_h^b is added to the m-set, the new m-set is used to compare with other transactions in the n-set with lower priority than s_h^b. Hence, if one of the transactions in the n-set, s_d^g, is of lower priority than s_h^b and conflicts with s_h^b, it will remain in the n-set.

The choice of the new transaction from the n-set depends on the original priority of transactions (hence the word “PCM” in the name of the algorithm). Thus, the algorithm avoids interrupting an already executing transaction to reduce its retry cost. In the meanwhile, it tries to avoid delaying the highest priority transaction in the n-set when it is time to select a new one to commit, even if the highest priority transaction arrives after other lower priority transactions in the n-set.

5.2.1 Properties

Claim 17 *Transactions scheduled under PNF do not suffer from transitive retry.*

Proof 17 *The proof is by contradiction. Assume that a transaction s_i^k is retrying because of a higher priority transaction s_j^l, which in turn is retrying because of another higher priority transaction s_z^h. Assume that s_i^k and s_z^h do not conflict, yet, s_i^k is transitively retrying because of s_z^h. Note that s_z^h and s_j^l cannot exit together in the m-set as they have common objects. But they both can exist in the n-set, as they both can conflict with other* executing transactions. *We have three cases:*

Case 1: Assume that s_z^h is an executing transaction. This means that s_j^l is in the n-set. When s_i^k arrives, by the definition of PNF, it will be compared with the m-set, which contains s_z^h. Now, it will be found that s_i^k does not conflict with s_z^h. Also, by the definition of PNF, s_i^k is not compared with transactions in the n-set. When it newly arrives, priorities of n-set transactions are lower than any normal priority. Therefore, as s_i^k does not conflict with any other executing transaction, it joins the m-set and becomes an executing transaction. *This contradicts the assumption that s_i^k is transitively retrying because of s_z^h.*

Case 2: Assume that s_z^h is in the n-set, while s_j^l is an executing transaction. When s_i^k arrives, it will conflict with s_j^l and joins the n-set. Now, s_i^k retries due to s_j^l, and not s_z^h. When s_j^l commits, the n-set is traversed from the highest priority transaction to the lowest one: if s_z^h does not conflict with any other executing transaction and there are available processors, s_z^h becomes an executing transaction. When s_i^k is compared with the m-set, it is found that it does not conflict with s_z^h. Additionally, if it also does not conflict with any other executing transaction and there are available processors, then s_i^k becomes an executing transaction. This means that s_i^k and s_z^h are executing concurrently, which violates the assumption of transitive retry.

Case 3: Assume that s_z^h and s_j^l both exist in the n-set. When s_i^k arrives, it is compared with the m-set. If s_i^k does not conflict with any executing transactions and there are available processors, then s_i^k becomes an executing transaction. Even though s_i^k has common objects with s_j^l, s_i^k is not compared with s_j^l, which is in the n-set. If s_i^k joins the n-set, it is because, it conflicts with one or more executing transactions, not because of s_z^h, which violates the transitive retry assumption. If the three transactions s_i^k, s_j^l and s_z^h exist in the n-set, and s_z^h is chosen as a new executing transaction, then s_j^l remains in the n-set. This leads to Case 1. If s_j^l is chosen, because s_z^h conflicts with another executing transaction and s_j^l does not, then this leads to Case 2. Claim follows.

Claim 18 *The* first access *property of PNF prevents transitive retry.*

Proof 18 *The proof is by contradiction. Assume that the retry cost of transactions in the absence of the first access property is the same as when first access exists. Now, assume that*

PNF is devoid of the first access property. This means that executing transactions can be aborted.

Assume three transactions s_i^k, s_j^l, and s_z^h, where s_z^h's priority is higher than s_j^l's priority, and s_j^l's priority is higher than s_i^k's priority. Assume that s_j^l conflicts with both s_i^k and s_z^h. s_i^k and s_z^h do not conflict together. If s_i^k arrives while s_z^h is an executing transaction and s_j^l exists in the n-set, then s_i^k becomes an executing transaction itself while s_j^l is retrying. If s_i^k did not commit at least when s_z^h commits, then s_j^l becomes an executing transaction. Due to the lack of the first access property, s_j^l will cause s_i^k to retry. So, the retry cost for s_i^k will be $len(s_z^h + s_j^l)$. This retry cost for s_i^k is the same if it had been transitively retrying because of s_z^h. This contradicts the first assumption. Claim follows.

From Claims 17 and 18, PNF does not increase the retry cost of multi-object transactions. However, this is not the case for ECM and RCM as shown by Claim 16.

Claim 19 *Under PNF, any job τ_i^x is not affected by the retry cost in any other job τ_j^l.*

Proof 19 *As explained in Section 4, PNF assigns a temporary priority of -1 to any job that includes a retrying transaction. So, retrying transactions have lower priority than any other normal priority. When τ_i^x is released and τ_j^l has a retrying transaction, τ_i^x will have a higher priority than τ_j^l. Thus, τ_i^x can run on any available processor while τ_j^l is retrying one of its transactions. Claim follows.*

5.3 Retry Cost under PNF

We now derive an upper bound on the retry cost of any job τ_i^x under PNF during an interval $L \leq T_i$. Since all tasks are sporadic (i.e., each task τ_i has a minimum period T_i), T_i is the maximum study interval for each task τ_i.

Claim 20 *Assume two conflicting transactions s_i^k and s_j^l. Under PNF, the maximum retry cost suffered by s_i^k due to s_j^l is $len(s_j^l)$.*

Proof 20 *By PNF's definition, s_i^k cannot have started before s_j^l. Otherwise, s_i^k would have been an executing transaction and s_j^l cannot abort it. So, the earliest release time for s_i^k would have been just after s_j^l starts execution. Then, s_i^k would have to wait until s_j^l commits. Claim follows.*

Claim 21 *The retry cost for any job τ_i^x due to conflicts between its transactions and transactions of other jobs under PNF during an interval $L \leq T_i$ is upper bounded by:*

$$RC(L) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^l}(\theta)} len\left(\bar{s_j^l}(\theta) \right) \right) \right) \tag{5.1}$$

where $\bar{s_j^l}(\theta)$ is the same as $s_j^l(\theta)$ except for the following difference: if $\bar{s_j^l}$ accesses multiple objects in θ_i, then $\bar{s_j^l}$ is included only once in the last summation (i.e., $\bar{s_j^l}$ is not repeated for each shared object with s_i^k).

Proof 21 *Consider a transaction s_i^k belonging to job τ_i^x. Under PNF, higher priority transactions than s_i^k can become preempted transaction before s_i^k. A lower priority transaction s_v^f can also become an executing transaction before s_i^k. This happens when s_i^k conflicts with any executing transaction while s_v^f does not. The worst case scenario for s_i^k occurs when s_i^k has to wait in the n-set, while all other conflicting transactions with s_i^k are chosen to be preempted transaction. Let $\bar{s_j^l}$ accesses multiple objects in θ_i. If $\bar{s_j^l}$ is an executing transaction, then $\bar{s_j^l}$ will not repeat itself for each object it accesses. Besides, $\bar{s_j^l}$ will finish before s_i^k starts execution. Consequently, $\bar{s_j^l}$ will not conflict with s_i^{k+1}. This means that an executing transaction can force no more than one transaction in a given job to retry. This is why $\bar{s_j^l}$ is included only once in (5.1) for all shared objects with s_i^k.*

The maximum number of jobs of any task τ_j that can interfere with τ_i^x during interval L is $\left\lceil \frac{L}{T_j} \right\rceil + 1$. From the previous observations and Claim 20, Claim follows.

Claim 22 *The blocking time for a job τ_i^x due to lower priority jobs, during an interval $L \leq T_i$, is upper bounded by:*

$$D(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h} \right) \right) \right\rfloor \tag{5.2}$$

where $D(\tau_i^x)$ is the blocking time suffered by τ_i^x due to lower priority jobs. $\bar{\tau_j^l} = \{\tau_j^l : p_j^l < p_i^x\}$ and $\ddot{s_j^h} = \{s_j^h : s_j^h$ does not conflict with any $s_i^k\}$. During this blocking time, all processors are unavailable for τ_i^x.

Proof 22 *Under PNF, preempted transaction are non-preemptive. So, lower priority preempted transaction can delay a higher priority job τ_i^x if no other processors are available. Lower priority executing transactions can be conflicting or non-conflicting with any transaction in τ_i^x. They also can exist when τ_i^x is newly released, or after that. So, we have the following cases:*

Lower priority conflicting transactions after τ_i^x is released: *This case is already covered by the retry cost in (5.1).*

Lower priority conflicting transactions when τ_i^x is newly released: *Each lower priority conflicting transaction s_j^h will delay τ_i^x for $len(s_j^h)$. The effect of s_j^h is already covered by (5.1). Besides, (5.1) does not divide the retry cost by m as done in (5.2). Thus, the worst case scenario requires inclusion of s_j^h in (5.1), and not in (5.2).*

Lower priority non-conflicting transactions when τ_i^x is newly released: *τ_i^x is delayed if there are no available processors for it. Otherwise, τ_i^x can run in parallel with these non-conflicting lower priority transactions. Each lower priority non-conflicting transaction $\ddot{s}_j^h$ will delay τ_i^x for $len(\ddot{s}_j^h)$.*

Lower priority non-conflicting transactions after τ_i^x is released: *This situation can happen if τ_i^x is retrying one of its transactions s_i^k. So, τ_i^x is assigned a priority of -1. τ_i^x can be preempted by any other job. When s_i^k is checked again to be an executing transaction, all processors may be busy with lower priority non-conflicting transaction and/or higher priority jobs. Otherwise, τ_i^x can run in parallel with these lower priority non-conflicting transactions. The effect of higher priority jobs is included by Claims 23, 24.*

Each lower priority non-conflicting transaction $\ddot{s}_j^h$ will delay τ_i^x for $len(\ddot{s}_j^h)$.

From the previous cases, lower priority non-conflicting transactions act as if they were higher priority jobs interfering with τ_i^x. So, the blocking time can be calculated by the interference workload given by Theorem 1 in [8]. Claim follows.

Claim 23 *Assume that PNF is used with the G-EDF scheduler. The response time of a job τ_i^x, during an interval $L \leq T_i$, is upper bounded by:*

$$R_i^{up} = c_i + RC(L) + D_{edf}(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} W_{ij}(R_i^{up}) \right\rfloor \tag{5.3}$$

where $RC(L)$ is calculated by (5.1). $D_{edf}(\tau_i^x)$ is the same as $D(\tau_i^x)$ defined in (5.2). However, for G-EDF systems. $D_{edf}(\tau_i^x)$ is calculated as:

$$D_{edf}(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau}_j^l} \begin{cases} 0 & , L \leq T_i - T_j \\ \sum_{\forall \ddot{s}_j^h} len\left(\ddot{s}_j^h\right) & , L > T_i - T_j \end{cases} \right\rfloor \tag{5.4}$$

and $W_{ij}(R_i^{up})$ is calculated by (3) in [31].

Proof 23 *Response time for τ_i^x is calculated by (3) in [31] with the addition of blocking time defined by Claim 22. G-EDF uses absolute deadlines for scheduling. This defines which jobs of the same task can be of lower priority than τ_i^x, and which will not. Any instance τ_j^h,* *released between $r_i^x - T_j$ and $d_i^x - T_j$, will be of higher priority than τ_i^x. Before $r_i^x - T_j$, τ_j^h would have finished before τ_i^x is released. After $d_i^x - T_j$, d_j^h would be greater than d_i^x. Thus, τ_j^h will be of lower priority than τ_i^x. So, during T_i, there can be only one instance τ_j^h of τ_j with lower priority than τ_i^x. τ_j^h is released between $d_i^x - T_j$ and d_i^x. Consequently, during $L < T_i - T_j$, no existing instance of τ_j is of lower priority than τ_i^x. Hence, 0 is used in the first case of (5.4). But if $L > T_i - T_j$, there can be only one instance τ_j^h of τ_j with lower priority than τ_i^x. Hence, $\left\lceil \frac{L}{T_i} \right\rceil + 1$ in (5.2) is replaced with 1 in the second case in (5.4). Claim follows.*

Claim 24 *Assume that PNF is used with the G-RMA scheduler. Response time of job τ_i^x during an interval $L \leq T_i$ is upper bounded by:*

$$R_i^{up} = c_i + RC(L) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i, p_j > p_i} W_{ij}(R_i^{up}) \right\rfloor \tag{5.5}$$

where $RC(L)$ is calculated by (5.1), $D(\tau_i^x)$ is calculated by (5.2), and $W_{ij}(R_i^{up})$ is calculated by (2) in [31].

Proof 24 *The proof is the same as for Claim 23, except that G-RMA assigns static priorities for tasks. Hence, (5.2) can be used directly for calculating $D(\tau_i^x)$ without modifications. Claim follows.*

5.4 Comparison between PNF and Other Synchronization Techniques

We now (formally) compare the schedulability of G-EDF (G-RMA) with PNF against ECM, RCM, LCM and lock-free synchronization [25, 30, 31]. Such a comparison will reveal when PNF outperforms others. Toward this, we compare the total utilization under G-EDF (G-RMA)/PNF, with that under the other synchronization methods. Inflated execution time of each method, which is the sum of the worst-case execution time of the task and its retry cost, is used in the utilization calculation of each task.

By Claim 22, no processor is available for τ_i^x during the blocking time. As each processor is busy with some other job than τ_i^x, $D(\tau_i^x)$ is not added to the inflated execution time of τ_i^x. Hence, $D(\tau_i^x)$ is not added to the utilization calculation of τ_i^x.

Let $RC_A(T_i)$ denote the retry cost of any τ_i^x using the synchronization method A during T_i. Let $RC_B(T_i)$ denote the retry cost of any τ_i^x using synchronization method B during T_i.

Then, schedulability of A is comparable to B if:

$$\begin{aligned}\sum_{\forall \tau_i} \frac{c_i + RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{c_i + RC_B(T_i)}{T_i} \\ \therefore \sum_{\forall \tau_i} \frac{RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{RC_B(T_i)}{T_i}\end{aligned} \tag{5.6}$$

As described in Section 5.1, the set of common objects needs to be extended under ECM and RCM. Toward this, we introduce a few additional notions. Let θ_i^{ex} be an extended set of distinct objects that contains all objects in θ_i. Thus, θ_i^{ex} contains all objects accessed by τ_i. θ_i^{ex} can also contain other objects that can cause any transaction in τ_i to retry as explained in Section 5.1. Thus, θ_i^{ex} may contain objects not accessed by τ_i. γ_i^{ex} is an extended set of tasks that access any object in θ_i^{ex}. Therefore, γ_i^{ex} contains at least all tasks in γ_i.

There are two sources of retry cost for any τ_i^x under ECM, RCM, LCM and lock-free. First is due to conflict between τ_i^x's transactions and transactions of other jobs. This is denoted as RC. Second is due to the preemption of any transaction in τ_i^x due to the release of a higher priority job τ_j^h. This is denoted as RC_{re}. Retry due to the release of higher priority jobs do not occur under PNF, because executing transactions are non-preemptive. It is up to the implementation of the contention manager to safely avoid RC_{re}. Here, we assume that ECM, RCM and LCM do not avoid RC_{re}. Thus, we introduce RC_{re} for ECM, RCM and LCM first before comparing PNF with other synchronization techniques.

Claim 25 *Under ECM and G-EDF/LCM the total retry cost suffered by all transactions in any τ_i^x during an interval $L \leq T_i$ is upper bounded by:*

$$RC_{to}(L) = RC(L) + RC_{re}(L) \tag{5.7}$$

where $RC(L)$ is the retry cost resulting from conflict between transactions in τ_i^x and transactions of other jobs. $RC(L)$ is calculated by (15) in [31] for ECM and (5) in [30] for G-EDF/LCM. γ_i and θ_i are replaced with γ_i^{ex} and θ_i^{ex}, respectively. $RC_{re}(L)$ is the retry cost resulting from the release of higher priority jobs, which preempt τ_i^x. $RC_{re}(L)$ is given by:

$$RC_{re}(L) = \sum_{\forall \tau_j \in \zeta_i} \begin{cases} \left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} & , L \leq T_i - T_j \\ \\ \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{i_{max}} & , L > T_i - T_j \end{cases} \tag{5.8}$$

where $\zeta_i = \{\tau_j : (\tau_j \neq \tau_i) \wedge (D_j < D_i)\}$.

Proof 25 *Two conditions must be satisfied for any τ_j^l to be able to preempt τ_i^x under G-EDF: $r_i^x < r_j^l < d_i^x$, and $d_j^l \leq d_i^x$. Without the first condition, τ_j^l would have been already*

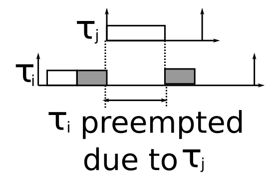


Figure 5.1: Transactional retry due to release of higher priority tasks

released before τ_i^x. Thus, τ_j^l will not preempt τ_i^x. Without the second condition, τ_j^l will be of lower priority than τ_i^x and will not preempt it. If $D_j \geq D_i$, then there will be at most one instance τ_j^l with higher priority than τ_i^x. τ_j^l must have been released at most at r_i^x, which violates the first condition. The other instance τ_j^{l+1} would have an absolute deadline greater than d_i^x. This violates the second condition. Hence, only tasks with shorter relative deadline than D_i are going to be considered. These jobs are grouped in ζ_i.

The total number of released instances of τ_j during any interval $L \leq T_i$ is $\left\lceil \frac{L}{T_i} \right\rceil + 1$. The "carried-in" jobs (i.e., each job released before r_i^x and has an absolute deadline before d_i^x [8]) are discarded as they violate the first condition. The "carried-out" jobs (i.e., each job released after r_i^x and has an absolute deadline after d_i^x [8]) are also discarded because they violate the second condition. Thus, the number of considered higher priority instances of τ_j during the interval $L \leq T_i - T_j$ is $\left\lceil \frac{L}{T_j} \right\rceil$. The number of considered higher priority instances of τ_j during interval $L > T_i - T_j$ is $\left\lfloor \frac{T_i}{T_j} \right\rfloor$.

The worst RC_{re} for τ_i^x occurs when τ_i^x is always interfered at the end of execution of its longest atomic section, $s_{i_{max}}$. τ_i^x will have to retry for $len(s_{i_{max}})$, as shown in Figure 5.1. The total retry cost suffered by τ_i^x is the combination of RC and RC_{re}. Claim follows.

Claim 26 *Under RCM and G-RMA/LCM, the total retry cost suffered by all transactions in any τ_i^x during an interval $L \leq T_i$ is upper bounded by:*

$$RC_{to}(L) = RC(L) + RC_{re}(L) \tag{5.9}$$

where $RC(L)$ and $RC_{re}(L)$ are defined in Claim 25. $RC(L)$ is calculated by (16) in [31] for RCM, and (8) in [30] for G-RMA/LCM. $RC_{re}(L)$ is calculated by:

$$RC_{re}(L) = \sum_{\forall \tau_j \in \zeta_i^*} \left(\left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} \right) \tag{5.10}$$

where $\zeta_i^* = \{\tau_j : p_j > p_i\}$.

Proof 26 *The proof is the same as that for Claim 25, except that G-RMA uses static priority. Thus, the carried-out jobs will be considered in the interference with τ_i^x. The carried-in jobs are still not considered because they are released before r_i^x. Claim follows.*

Claim 27 *Consider lock-free synchronization. Let $r_{i_{max}}$ be the maximum execution cost of a single iteration of any retry loop of τ_i. RC_{re} under G-EDF with lock-free synchronization is calculated by (5.8), where $s_{i_{max}}$ is replaced by $r_{i_{max}}$. RC_{re} under G-RMA with lock-free synchronization is calculated by (5.10), where $s_{i_{max}}$ is replaced by $r_{i_{max}}$.*

Proof 27 *The interference pattern of higher priority jobs to lower priority jobs is the same in ECM, G-EDF/LCM and G-EDF with lock-free synchronization. The pattern is also the same in RCM, G-RMA/LCM and G-RMA with lock-free. Claim follows.*

5.4.1 PNF versus ECM

Claim 28 *The schedulability of PNF with G-EDF is better or equal to the schedulability of ECM when conflicting atomic sections have equal lengths.*

Proof 28 *Substitue $RC_A(T_i)$ and $RC_B(T_i)$ in (5.6) with (5.1) and (5.7), respectively. Let $\theta_i^{ex} = \theta_i + \theta_i^*$, where θ_i^* is the set of objects not accessed directly by τ_i but can cause transactions in τ_i to retry due to transitive retry. Let $\gamma_i^{ex} = \gamma_i + \gamma_i^*$, where γ_i^* is the set of tasks that access objects in θ_i^*.*

Let:

$$
\begin{aligned}
g(\tau_i) \;&=\; \left(\sum_{\forall \tau_j \in \gamma_i^*} \sum_{\theta \in \theta_i^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^{\bar{k}}(\theta)} len\Big(s_j^{\bar{k}}(\theta) \right.\right. \\
&+ \left.\left.\; s_{max}^*(\theta)\Big)\right)\right) + RC_{re}(T_i)
\end{aligned}
$$

where RC_{re} is given by (5.8). Let:

$$\eta_1(\tau_i) = \sum_{\forall \tau_j \in \gamma_i} \sum_{\forall \theta \in \theta_i} \left(\sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta)\right) \right)$$

$$\eta_2(\tau_i) = \sum_{\forall \tau_j \in \gamma_i} \sum_{\forall \theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_{max}^j(\theta)\right) \right)$$

and

$$\eta_3(\tau_i) = \sum_{\forall \tau_j \in \gamma_i} \sum_{\forall \theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta)\right) \right)$$

By substitution of $g(\tau_i)$, $\eta_1(\tau_i)$, and $\eta_2(\tau_i)$, and subtraction of $\sum_{\forall\tau_i}\frac{\eta_3(\tau_i)}{T_i}$ from both sides of (5.6), we get:

$$\sum_{\forall\tau_i}\frac{\eta_1(\tau_i)}{T_i} \le \sum_{\forall\tau_i}\frac{\eta_2(\tau_i)+g(\tau_i)}{T_i} \tag{5.11}$$

From (5.11), we note that by keeping every $len(s_j^{\bar{k}}(\theta)) \le len(s_{max}^j(\theta))$ for each τ_i, $\tau_j \in \gamma_i$, and $\theta \in \theta_i$, (5.11) holds. Because of the dynamic priority of G-EDF, $s_{max}^j(\theta)$ can belong to any task other than τ_j. Assume four jobs τ_a^b, τ_c^d, τ_e^f, and τ_g^h with a common object θ. Let $s_{max}(\theta) = s_{g_{max}}(\theta)$. When τ_a^b is the checked τ_i by (5.11), any $s_c^x(\theta)$ and $s_e^y(\theta)$ will be less or equal to $s_{g_{max}}(\theta)$. But $s_{e_{max}}(\theta)$ should also be smaller or equal to either $s_{a_{max}}(\theta)$ or $s_{c_{max}}(\theta)$ or $s_{g_{max}}(\theta)$. Thus, there must be at least two equal maximum-length atomic sections in different tasks that access θ. By generalizing the previous concept to every τ_i, $\tau_j \in \gamma_i$, and $\theta \in \theta_i$, claim follows.

5.4.2 PNF versus RCM

Claim 29 *The schedulability of PNF with G-RMA is better or equal to the schedulability of RCM when a large number of tasks are heavily conflicting together. Increasing atomic section length of higher priority tasks improves schedulability of PNF compared with G-RMA/LCM schedulabiltiy.*

Proof 29 *Let $\theta_i^{ex} = \theta_i + \theta_i^*$ and $\gamma_i^{ex} = \gamma_i + \gamma_i^*$, as defined in the proof of Claim 28. Substitute $RC_A(T_i)$ and $RC_B(T_i)$ in (5.6) with (5.1) and (5.9), respectively. Let*

$$\begin{aligned} g(\tau_i) &= RC_{re}(T_i) + \Bigg(\sum_{\forall\tau_j\in(\gamma_i^*\cap\zeta_i^*)}\sum_{\forall\theta\in\theta_i^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\times \\ &\qquad \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta)+s_{max}^j(\theta)\right)\Bigg)\end{aligned}$$

where RC_{re} and ζ_i^* are defined by (5.10). Let $\gamma_i = \zeta_i^* \cup \bar{\zeta}_i$, where $\bar{\zeta}_i = \{\tau_j : (\tau_j \neq \tau_i) \wedge (p_j < p_i)\}$, thus $\zeta_i^* \cap \bar{\zeta}_i = \phi$.

Let:

$$\eta_1(\tau_i) = \sum_{\forall\tau_j\in(\gamma_i\cap\zeta_i^*)}\sum_{\forall\theta\in\theta_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta)\right)\right)$$

$$\eta_2(\tau_i) = \sum_{\forall\tau_j\in(\gamma_i\cap\bar{\zeta}_i)}\sum_{\forall\theta\in\theta_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta)\right)\right)$$

and

$$\eta_3(\tau_i) = \sum_{\forall \tau_j \in (\gamma_i \cap \zeta_i^*)} \sum_{\forall \theta \in \theta_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \times \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta) + s_{max}^j(\theta) \right) \right)$$

By substitution of $g(\tau_i)$, $\eta_1(\tau_i)$, $\eta_2(\tau_i)$, and $\eta_3(\tau_i)$ in (5.6), we get:

$$\sum_{\forall \tau_i} \frac{\eta_1(\tau_i) + \eta_2(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_3(\tau_i) + g(\tau_i)}{T_i} \tag{5.12}$$

When tasks with deadlines equal to periods are scheduled with G-RMA, $T_j > T_i$ if $p_j < p_i$. So, for each $\tau_j \in \bar{\zeta}_i$, $\left\lceil \frac{T_i}{T_j} \right\rceil = 1$. Then:

$$\eta_2(\tau_i) = 2 \sum_{\forall \tau_j \in (\gamma_i \cap \bar{\zeta}_i)} \sum_{\forall \theta \in \theta_i} \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta) \right) \tag{5.13}$$

Let:

$$\eta_4(\tau_i) = \sum_{\forall \tau_j \in (\gamma_i \cap \zeta_i^*)} \sum_{\forall \theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_{max}^j(\theta) \right)$$

By substitution of (5.13) and subtraction of $\sum_{\forall \tau_i} \frac{\eta_1(\tau_i)}{T_i}$ from both sides of (5.12), we get:

$$2 \sum_{\forall \tau_i} \frac{\eta_2(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_4(\tau_i) + g(\tau_i)}{T_i} \tag{5.14}$$

From (5.14), we note that when higher priority jobs increasingly conflict with lower priority jobs, (5.14) tends to hold. This occurs when the number of conflicting tasks, their job instances, and their shared objects increases. When the number of shared objects among tasks increases, $g(\tau_i)$ also increases. This allows (5.14) to hold. Claim follows.

5.4.3 PNF versus G-EDF/LCM

Claim 30 *Schedulability of PNF/EDF is equal or better than schedulability of G-EDF/LCM if lengths of conflicting atomic sections are approximately equal and all αs approach unity.*

Proof 30 *Assuming $\eta_1(\tau_i)$ and $\eta_3(\tau_i)$ are the same as defined in proof of Claim 28. Let*

$$
\begin{aligned}
g(\tau_i) &= \Bigg(\sum_{\forall \tau_j \in \gamma_i^*} \sum_{\theta \in \theta_i^*} \Bigg(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^{\bar{k}}(\theta)} len\Big(s_j^{\bar{k}}(\theta) \\
&+ \alpha_{max}^{ji} s_{max}^*(\theta) \Big) \Bigg) \Bigg) + RC_{re}(T_i)
\end{aligned}
$$

$$
\eta_2(\tau_i) = \sum_{\forall \tau_j \in \gamma_i} \sum_{\forall \theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(\alpha_{max}^{jl} s_{max}^j(\theta) \right) \right)
$$

Following the same steps in proof of Claim 28, we get

$$
\sum_{\forall \tau_i} \frac{\eta_1(\tau_i)}{T_i} \le \sum_{\forall \tau_i} \frac{\eta_2(\tau_i) + g(\tau_i)}{T_i} \tag{5.15}
$$

Assuming $g(\tau_i)_{\forall \tau_i} \to 0$*, thus neglecting the effect of transitive retry and retry cost due to release of higher priority jobs. Let* $len(s_j^{\bar{k}}(\theta)) = s_{max}^j(\theta) = s$*, and* $\alpha_{max}^{jl} = \alpha_{max}^{iy} = 1$ *in (5.15), then schedulability of PNF/EDF equals schedulability of LCM/EDF if* $\left\lceil \frac{T_i}{T_j} \right\rceil = 1, \forall \tau_i, \tau_j$ *(which means equal periods for all tasks). If* $\left\lceil \frac{T_i}{T_j} \right\rceil > 1, \forall \tau_i, \tau_j$*, then schedulability of PNF/EDF is better than LCM/EDF. Schedulability of PNF/EDF becomes more better than schedulability of LCM/EDF if* $g(\tau_i)$ *is not zero. Claim follows.*

5.4.4 PNF versus G-RMA/LCM

Claim 31 *Schedulability performance of PNF is equal or better than schedulability performance of G-RMA/LCM if: 1) conflict effect of higher priority tasks to lower priority tasks increases. 2)Lengths of atomic sections increase as tasks' priorities increase. 3)* α*s approach unity.*

Proof 31 *Assume* $g(\tau_i)$*,* $\eta_1(\tau_i)$*,* $\eta_2(\tau_i)$ *are the same as proof of Claim 29. Let*

$$
\begin{aligned}
\eta_3(\tau_i) &= \sum_{\forall \tau_j \in (\gamma_i \cap \zeta_i^*)} \sum_{\forall \theta \in \theta_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \times \right. \\
&\left. \sum_{\forall s_j^{\bar{k}}(\theta)} len\left(s_j^{\bar{k}}(\theta) + \alpha_{max}^{jl} s_{max}^j(\theta) \right) \right)
\end{aligned}
$$

Following the setps of proof of Claim 29

$$\therefore \sum_{\forall \tau_i} \frac{\eta_1(\tau_i)+\eta_2(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_3(\tau_i)+g(\tau_i)}{T_i} \tag{5.16}$$

Let

$$\begin{aligned} \eta_4(\tau_i) &= \sum_{\forall \tau_j \in (\gamma_i \cap \zeta_i^*)} \sum_{\forall \theta \in \theta_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \right. \\ &\quad \left. \times \sum_{\forall \bar{s_j^k}(\theta)} len\left(\alpha_{max}^{jl} s_{max}^{j}(\theta)\right) \right) \end{aligned}$$

∴(5.16) becomes

$$2\sum_{\forall \tau_i} \frac{\eta_2(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_4(\tau_i)+g(\tau_i)}{T_i} \tag{5.17}$$

Assuming negligible effect of transitive retry and retry cost due to release of higher priority jobs ($g(\tau_i) \to 0$). (5.17) holds if: 1) contention from higher priority jobs to lower priority jobs increases because of the $\left\lceil \frac{T_i}{T_j} \right\rceil + 1$ term in the right hand side of (5.17). 2) αs approaches unity. 3) Lengths of atomic sections increase as priority increases. This makes $len(s_{max}^{j}(\theta))$ in the right hand side of (5.17) greater than $len(\bar{s_j^k}(\theta))$. Claim follows.

5.4.5 PNF versus lock-free

Lock-free synchronization [25, 31] accesses only one object. Thus, the number of accessed objects per transaction in PNF is limited to one. This allows us to compare the schedulability of PNF with the lock-free algorithm.

$RC_B(T_i)$ in (5.6) is replaced with:

$$\sum_{\forall \tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} r_{max} \right) + RC_{re}(T_i) \tag{5.18}$$

where $\beta_{i,j}$ is the number of retry loops of τ_j that access the same object as accessed by some retry loop of τ_i [25]. r_{max} is the maximum execution cost of a single iteration of any retry loop of any task [25]. $RC_{re}(T_i)$ is defined in Claim 27. Lock-free synchronization does not depend on priorities of tasks. Thus, (5.18) applies for both G-EDF and G-RMA systems.

Claim 32 *Let r_{max} be the maximum execution cost of a single iteration of any retry loop of any task [25]. Let s_{max} be the maximum transaction length in all tasks. Assume that* *each transaction under PNF accesses only one object for once. The schedulability of PNF with either G-EDF or G-RMA scheduler is better or equal to the schedulability of lock-free synchronization if $s_{max}/r_{max} \leq 1$.*

Proof 32 *The assumption in Claim 5.19 is made to enable a comparison between PNF and the lock-free technique. Let $RC_A(T_i)$ in (5.6) be replaced with (5.1) and $RC_B(T_i)$ be replaced with (5.18). To simplify comparison, (5.1) is upper bounded by:*

$$RC(T_i) = \sum_{\tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta^*_{i,j} s_{max} \right)$$

where $\beta^*_{i,j}$ is the number of times transactions in τ_j accesses shared objects with τ_i. Thus, $\beta^*_{i,j} = \beta_{i,j}$. Thus, (5.6) will be:

$$\sum_{\forall \tau_i} \frac{\sum_{\tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} s_{max} \right)}{T_i} \leq$$

$$\sum_{\forall \tau_i} \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} r_{max} + RC_{re}(\tau_i)}{T_i} \tag{5.19}$$

From (5.19), we note that if $s_{max} \leq r_{max}$, then (5.19) holds. Claim follows.

5.5 Conclusions

Transitive retry increases transactions' retry cost under ECM, RCM and LCM. PNF is designed to avoid transitive retry by non preempting transaction at execution. Aborted transactions are ordered in priority in *n_set* list. When an executing transaction commits, it triggers the highest priority transaction s_i^j in the *n_set* to check conflicts with executing transactions. s_i^j, in turn, triggers the second transaction in the *n_set*, and so on. Priority of aborted transactions is reduced to enable other tasks to execute. Thus, increasing processor usage. Executing transactions are not preempted due to release of higher priority jobs. In case of the other CMs, retry due to release of higher priority jobs depends on the CM implementation. On the negative side of PNF, higher priority jobs can be blocked by executing transactions belonging to lower priority jobs. Thus, increasing their response time. Schedulability under EDF/PNF is equal or better to ECM schedulability when lengths of atomic sections are are almost equal. RMA/PNF schedulability is equal or better than RCM in case of high conflict from higher priority jobs to lower priority ones. The previous conditions apply to schedulability comparison between PNF and LCM, in addition to increasing αs to unity. This is logical as LCM with G-EDF(G-RMA) acts as ECM(RCM) with $\alpha \to 1$. For schedulability of PNF to be equal or better than lock-free, the upper bound on s_{max}/r_{max}

is 1 instead of 0.5 under ECM and RCM. In the future, we are looking for combining PNF and LCM. This combination enables CM to act as PNF in case of highly transitive retry, and act as LCM otherwise. Thus, retry cost is reduced in all cases.

Chapter 6

Experiments

6.1 LCM, ECM and RCM

Having established LCM's retry and response time upper bounds, and the conditions under which it outperforms ECM, RCM, and lock-free synchronization, we now would like to understand how LCM's retry and response times in practice (i.e., on average) compare with that of competitor methods. Since this can only be understood experimentally, we implement LCM and the competitor methods and conduct experimental studies.

6.1.1 Experimental Setup

We used the ChronOS real-time Linux kernel [24] and the RSTM library [54]. We modified RSTM to include implementations of ECM, RCM, G-EDF/LCM, and G-RMA/LCM contention managers, and modified ChronOS to include implementations of G-EDF and G-RMA schedulers.

For the retry-loop lock-free implementation, we used a loop that reads an object and attempts to write to the object using a compare-and-swap (CAS) instruction. The task retries until the CAS succeeds.

We use an 8 core, 2GHz AMD Opteron platform. The average time taken for one write operation by RSTM on any core is $0.0129653375\mu s$, and the average time taken by one CAS-loop operation on any core is 0.0292546250 μs.

We used the periodic task set shown in Table 6.1. Each task runs in its own thread and has a set of atomic sections. Atomic section properties are probabilistically controlled (for experimental evaluation) using three parameters: the maximum and minimum lengths of any atomic section within the task, and the total length of atomic sections within any task. All task atomic sections access the same object, and do write operations on the object (thus,

Table 6.1: Task sets. (a) Task set 1: 5-task set; (b) Task set 2: 10-task set; (c) Task set 3: 12-task set

(a)

	$T_i(\mu s)$	$c_i(\mu s)$
τ_1	500000	150000
τ_2	1000000	227000
τ_3	1500000	410000
τ_4	3000000	299000
τ_5	5000000	500000

(b)

	$T_i(\mu s)$	$c_i(\mu s)$
τ_1	400000	75241
τ_2	750000	69762
τ_3	1200000	267122
τ_4	1500000	69863
τ_5	2400000	152014
τ_6	4000000	286301
τ_7	7500000	493150
τ_8	10000000	794520
τ_9	15000000	1212328
τ_{10}	20000000	1775342

(c)

	$T_i(\mu s)$	$c_i(\mu s)$
τ_1	400000	58195
τ_2	750000	53963
τ_3	1000000	206330
τ_4	1200000	53968
τ_5	1500000	117449
τ_6	2400000	221143
τ_7	3000000	290428
τ_8	4000000	83420
τ_9	7500000	380917
τ_{10}	10000000	613700
τ_{11}	15000000	936422
τ_{12}	20000000	1371302

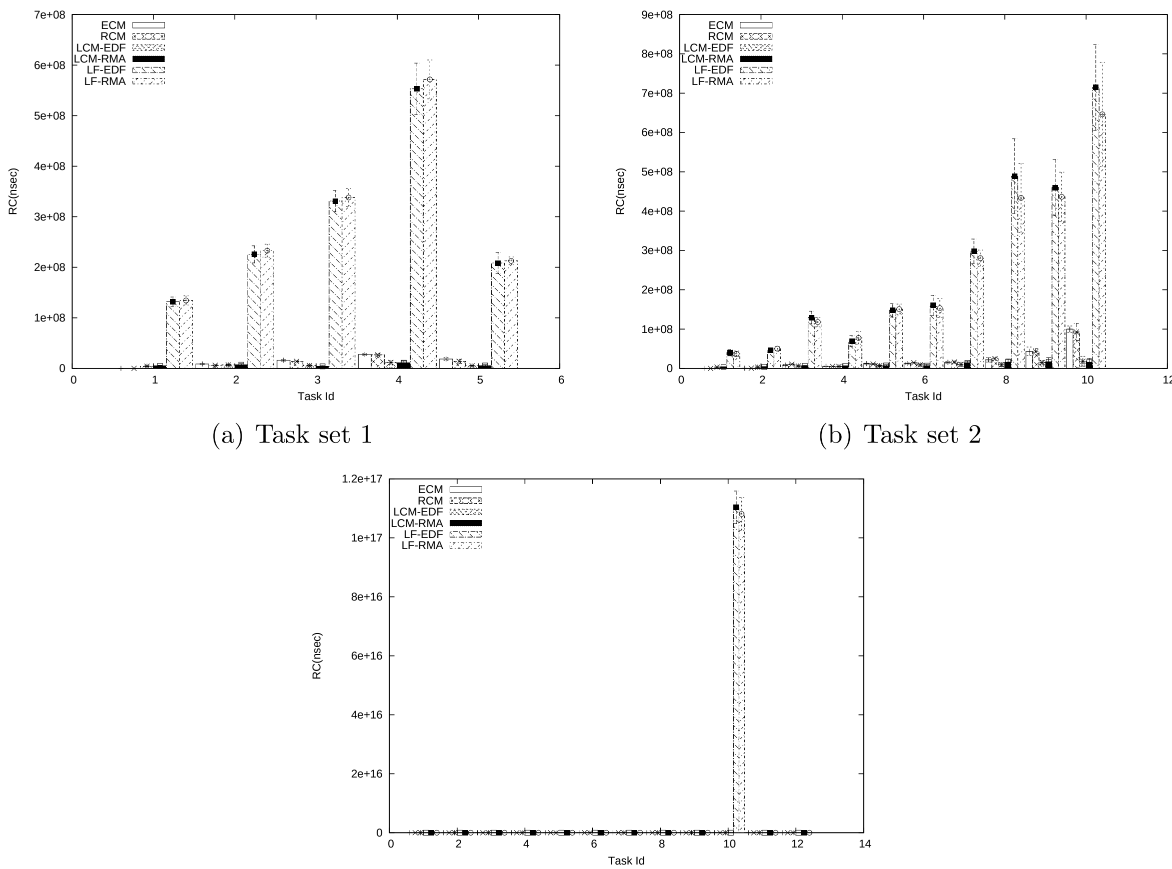
contention is the highest).

6.1.2 Results

Figure 6.1 shows the retry cost (RC) for each task in the three task sets in Table 6.1, where each task has a single atomic section of length equal to 0.2 of the task length. Each data point in the figure has a confidence level of 0.95. We observe that G-EDF/LCM and G-RMA/LCM achieve shorter or comparable retry cost than ECM and RCM. Since all tasks are initially released at the same time, and due to the specific nature of task properties, tasks with lower IDs somehow have higher priorities under the G-EDF scheduler. Note that tasks with lower IDs have higher priorities under G-RMA, since tasks are ordered in non-decreasing order of their periods.

Thus, we observe that G-EDF/LCM and G-RMA/LCM achieve comparable retry costs to ECM and RCM for some tasks with lower IDs. But when task ID increases, LCM — for both schedulers — achieves much shorter retry costs than ECM and RCM. This is because, higher priority tasks in LCM can be delayed by lower priority tasks, which is not the case for ECM and RCM. However, as task priority decreases, LCM, by definition, prevents higher priority tasks from aborting lower priority ones if a higher priority task interferes with a lower priority one after a specified threshold. In contrast, under ECM and RCM, lower priority tasks abort in favor of higher priority ones. G-EDF/LCM and G-RMA/LCM also achieve shorter retry costs than the retry-loop lock-free algorithm.

Figure 6.2 shows the response time of each task of the task sets in Table 6.1 with a confidence



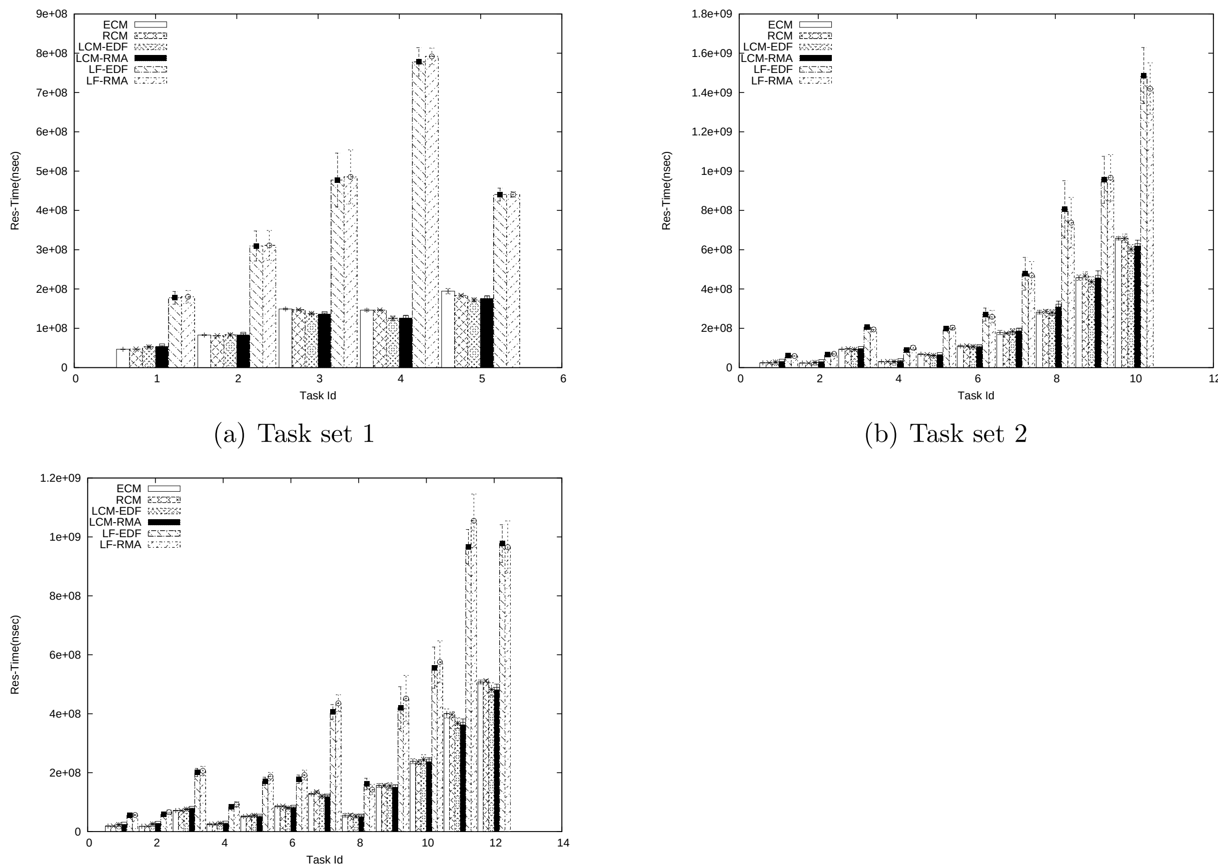
(a) Task set 1

(b) Task set 2

(c) Task set 3

Figure 6.1: Task retry costs under LCM and competitor synchronization methods

level of 0.95. (Again, each task's atomic section length is equal to half of the task length.) We observe that G-EDF/LCM and G-RMA/LCM achieve shorter response time than the retry-loop lock-free algorithm, and shorter or comparable response time than ECM and RCM.



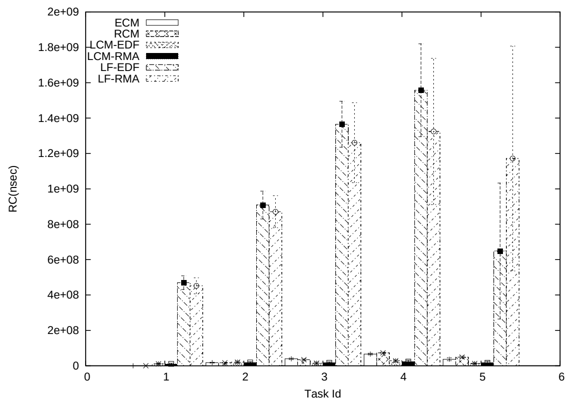
(a) Task set 1

(b) Task set 2

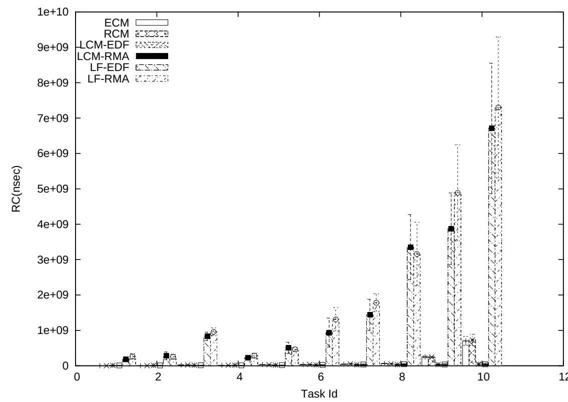
(c) Task set 3

Figure 6.2: Task response times under LCM and competitor synchronization methods

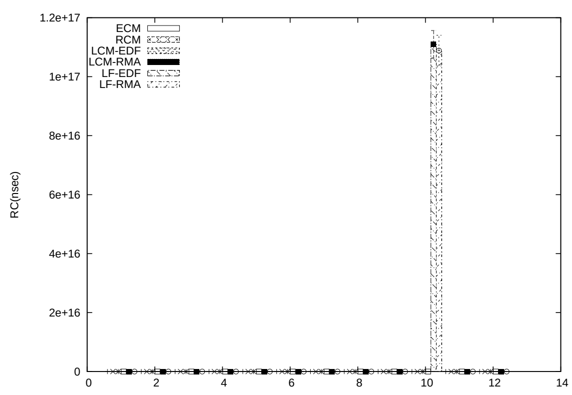
We repeated the experiments by varying the number and length of atomic sections. Due to space limitations, only a subset of the results are shown in Figures 6.3 to 6.6. (The complete set of results are shown in Appendix B of [29].) Each figure has three parameters x, y, and z in the label. x specifies the relative total length of all atomic sections to the length of the task. y specifies the maximum relative length of any atomic section to the length of the task. z specifies the minimum relative length of any atomic section to the length of the task. These figures show a consistent trend with previous results.



(a) Task set 1

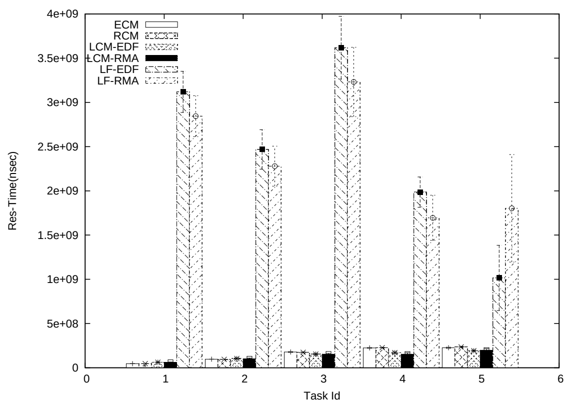


(b) Task set 2

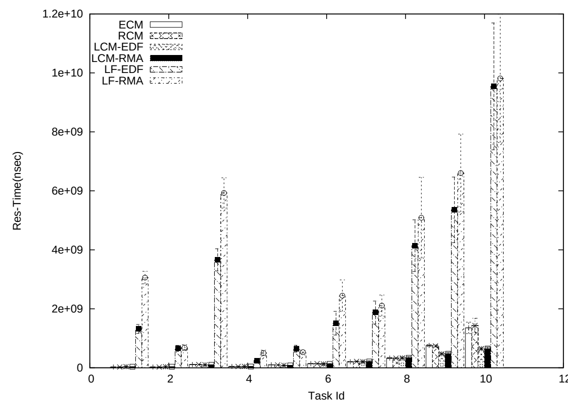


(c) Task set 3

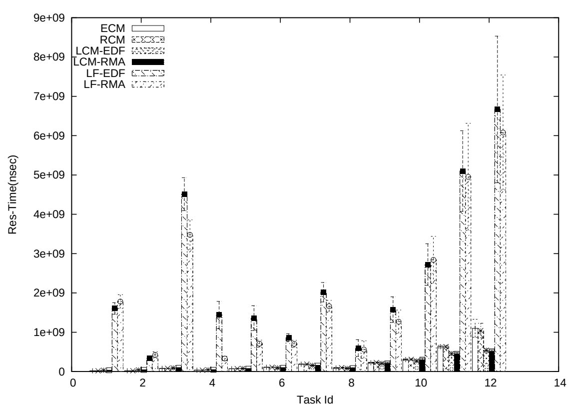
Figure 6.3: Task retry costs under LCM and competitor synchronization methods (0.5,0.2,0.2)



(a) Task set 1

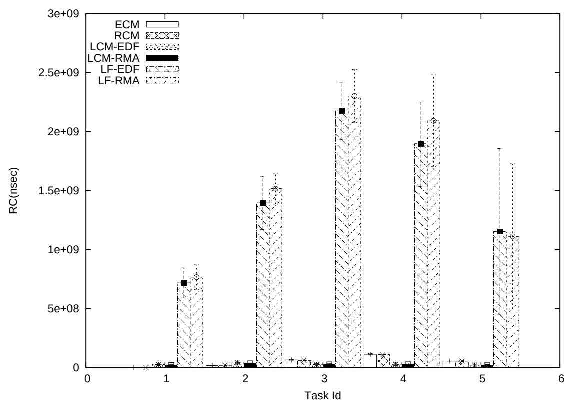


(b) Task set 2

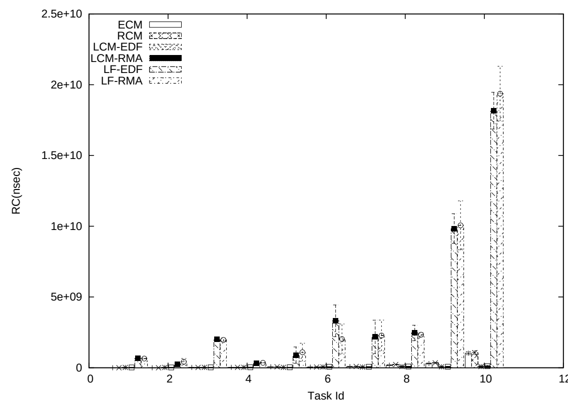


(c) Task set 3

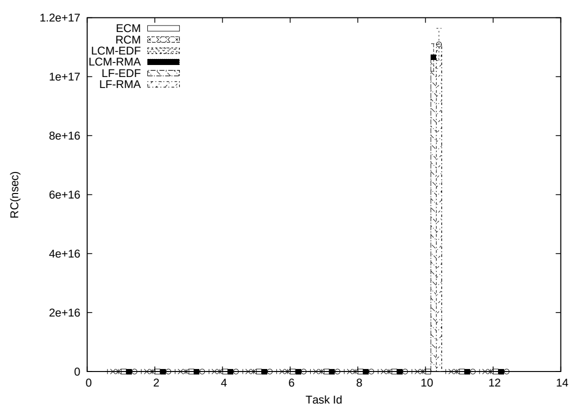
Figure 6.4: Task response times under LCM and competitor synchronization methods (0.5,0.2,0.2)



(a) Task set 1

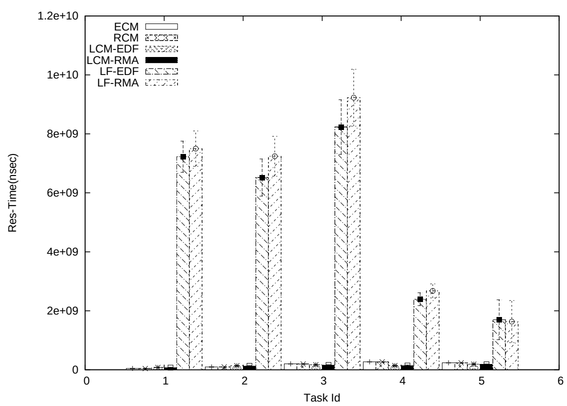


(b) Task set 2

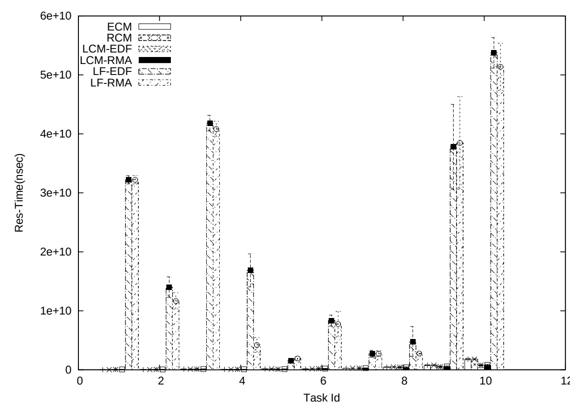


(c) Task set 3

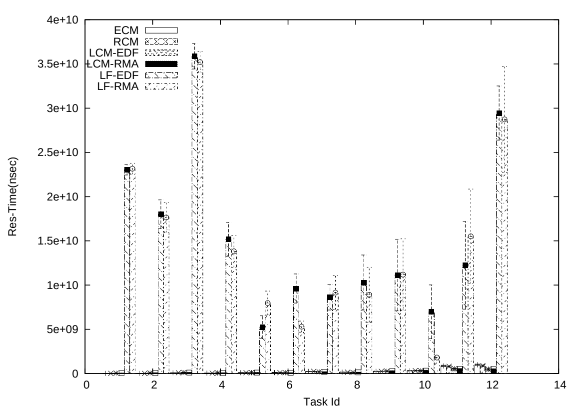
Figure 6.5: Task retry costs under LCM and competitor synchronization methods (0.8,0.5,0.2)



(a) Task set 1



(b) Task set 2



(c) Task set 3

Figure 6.6: Task response times under LCM and competitor synchronization methods (0.8,0.5,0.2)

6.2 PNF

Lock-free cannot handle more than object per atomic section. So, we compare retry cost of PNF against retry cost of other contention managers and lock-free in case of one object per transaction. Then, we compare retry cost of PNF against ECM, RCM and LCM in case of multiple objects per transactions. We used 3 sets of 4, 8 and 20 tasks. The structure of these tasks are shown in Table 6.2. The difficulty in testing with PNF is to incur transitive retry cases. Tasks are arranged in non-decreasing order of periods, and each task shares objects only with the previous and next tasks. Each task begins with an atomic section. Thus, increasing the opportunity of transitive retry.

Table 6.2: Task sets a) 4 tasks. b) 8 tasks. c) 20 tasks.

(a)

$P_i(\mu s)$	$c_i(\mu s)$
1000000	227000
1500000	410000
3000000	299000
5000000	500000

(b)

$P_i(\mu s)$	$c_i(\mu s)$
1500000	961000
1875000	175000
2500000	205000
3000000	129000
3750000	117000
5000000	269000
7500000	118000
15000000	609000

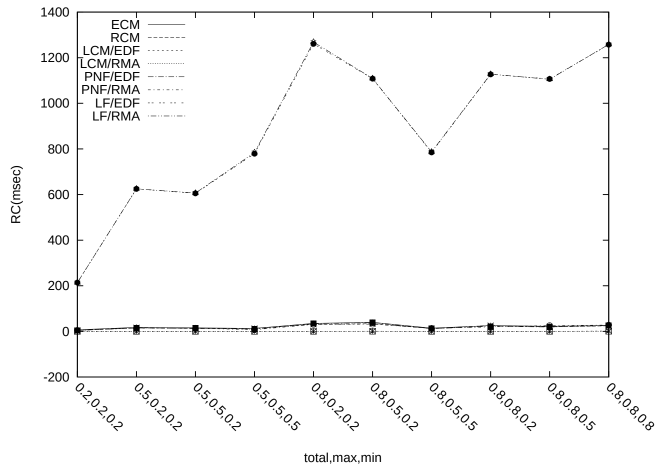
(c)

$P_i(\mu s)$	$c_i(\mu s)$
375000	9000
400000	8000
500000	8000
600000	14000
625000	375000
750000	19000
1000000	26000
1200000	17000
1250000	21000
1500000	33000
1875000	39000
2000000	43000
2500000	18000
3000000	90000
3750000	28000
5000000	126000
7500000	231000
10000000	407000
15000000	261000
30000000	369000
375000	8000
30000000	407000

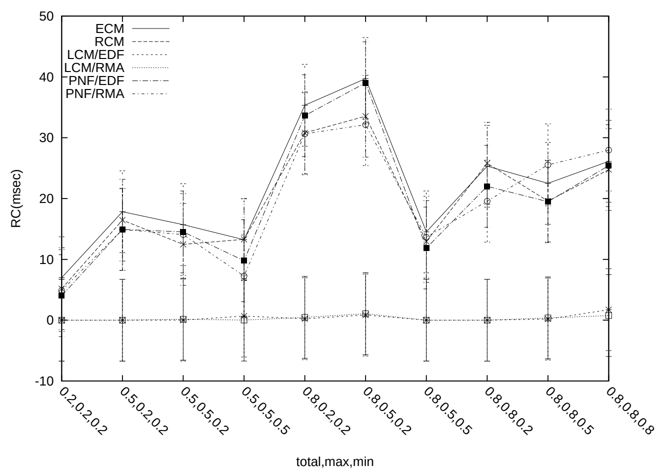
Figure 6.7(a) shows average retry cost under ECM, RCM, LCM, PNF and lock-free. Figure 6.7(b) shows average retry cost for only contention managers. Only one object per

transaction is shared in Figures 6.7(a) and 6.7(b). Lock-free is still the longest technique as it provides no conflict resolution. LCM (with both G-EDF and G-RMA) is better than the others. PNF (with G-EDF and G-RMA) approximates ECM and RCM because there is no transitive retry here.

Figure 6.8 shows average retry cost for three task sets in case of multiple objects per transaction. Each data point in the figure has a confidence level of 0.95. PNF (with G-EDF and G-RMA) achieves shorter or comparable retry cost than ECM, RCM and LCM. In Figure 6.8(c), retry costs are close. This happens because each task execution time is small as indicated in Table 6.2. Consequently, atomic sections have small length and highly unlikely to be triggered at the same time. Thus, contention, as well as, retry cost are low.

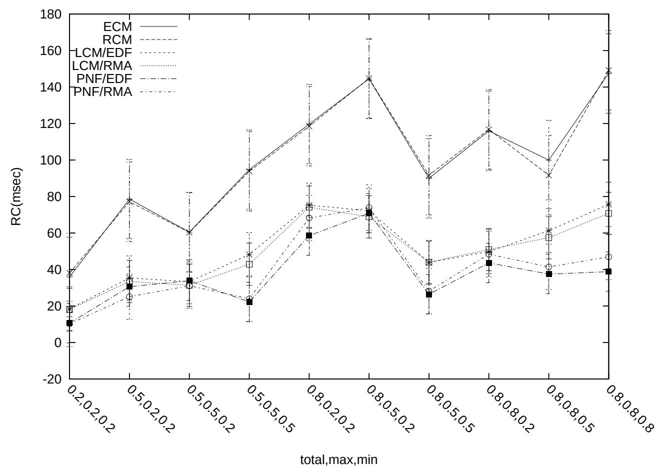


(a) ECM, RCM, LCM, PNF, Lock-Free

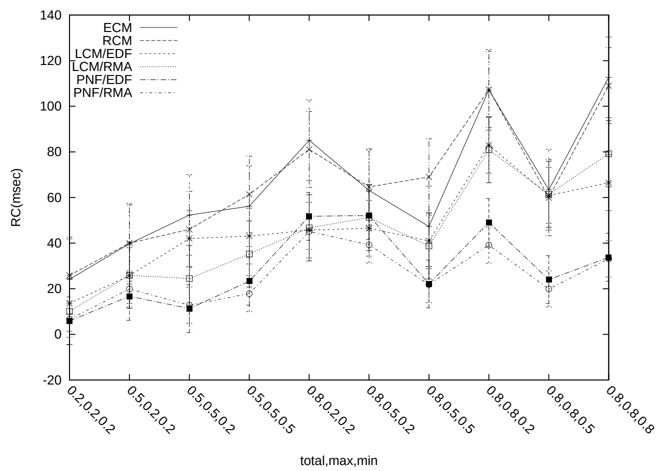


(b) ECM, RCM, LCM, PNF

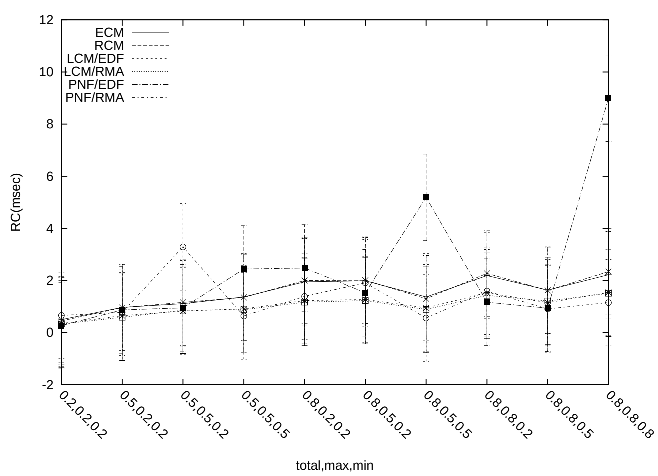
Figure 6.7: Average retry cost for 1 object per transaction for different values of total, maximum and minimum atomic section length under: a) all synchronization techniques. b) only contention managers.



(a) Task set 1



(b) Task set 2



(c) Task set 3

Figure 6.8: Average retry cost for different values of total, maximum and minimum atomic section length for: a) 4 tasks. b) 8 tasks. c) 20 tasks.

Chapter 7

Conclusions, Contributions, and Proposed Post Preliminary-Exam Work

we consider STM for concurrency control in multicore real-time software. Doing so will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy time constraints. Retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are scheduler-dependent). Thus, real-time CM is logical.

We investigate and design a number of real-time CMs. The first two CMs are directly based on dynamic and static priority of underlying tasks. Earliest Deadline-First CM with G-EDF scheduler (ECM) resolves conflicts based on absolute deadline of the underlying instances. Rate Monotonic Assignment with G-RMA scheduler (RCM) resolves conflicts based on period of underlying instances. We analyze retry cost and response time under ECM and RCM. We analytically and experimentally compare their schedulability against lock-free method.

ECM and RCM conserve the semantics of the underlying real-time scheduler. This conservative approach results in a maximum retry cost- for a single transaction due to another transaction- of double the maximum atomic section length among all tasks. So, another CM is developed to reduce this retry cost. Length-based CM (LCM) considers not only static/dynamic priority of underlying instance, but also length of the interfering transaction compared to remaining length of interfered transaction. LCM is used with G-EDF and G-RMA. Although it can reduce retry cost, but it suffers from priority inversion. By proper choice of different parameters, additional cost due to priority inversion can be kept lower than reduced retry cost. Thus, the net result will be lower response time for tasks using LCM with G-EDF/G-RMA. We analyze retry cost and response time of LCM. We analytically and experimentally compare LCM schedulability against ECM, RCM and lock-free.

ECM, RCM and LCM are affected by transitive retry. Transitive retry enforces a transaction to abort and retry due to another non-conflicting transaction. Transitive retry appears when multiple objects exist per transaction. So, we develop the Priority-based with Negative value and First access (PNF) contention manager. PNF avoids transitive retry and deals better with multiple objects than previous contention managers. PNF also tries to optimize processor usage by lowering priority of the job underlying retrying transaction. Thus, other jobs can proceed if there is no conflict. We upper bound retry cost and response time for PNF when used with G-EDF and G-RMA. Schedulability is compared between PNF on one side and ECM, RCM, LCM and lock-free on the other. We experimentally compare retry cost of PNF compared to other synchronization techniques.

7.1 Contribution

We design a number of contention managers that try to preserver real-time constraints besides data accuracy. Designing CMs is straightforward. The simplest logic is to keep the rational of the underlying real-time scheduler. This was shown in ECM and RCM. ECM allows transaction with earliest absolute deadline (dynamic priority) to commit first. RCM allows transaction with smallest period (fixed priority) to commit first. We derived upper bounds for retry cost and response time under both ECM and RCM. Lock-free schedulability was compared analytically and experimentally to schedulability of ECM and RCM. Under both ECM and RCM, a task incurs $2.s_{max}$ retry cost for each of its atomic sections due to a conflict with another task's atomic section. Retries under RCM and lock-free are affected by a larger number of conflicting task instances than under ECM. While task retries under ECM and lock-free are affected by all other tasks, retries under RCM are affected only by higher priority tasks.

STM and lock-free have similar parameters that affect their retry costs—i.e., the number of conflicting jobs and how many times they access shared objects. The s_{max}/r_{max} ratio determines whether STM is better or as good as lock-free. For ECM, this ratio cannot exceed 1, and it can be 1/2 for higher number of conflicting tasks. For RCM, for the common case, s_{max} must be 1/2 of r_{max}, and in some cases, s_{max} can be larger than r_{max} by many orders of magnitude.

We present Length-based contention manager (LCM) that is used with G-EDF and G-RMA. LCM tries to compromise between priority of transactions (which is priority of the underlying task), and remaining execution time of interfered transaction. As the remaining execution time of the interfered transaction decreases, it will be useless to abort it while it can shortly commit. To abort the interfered transaction or not, is determined by a α and ψ. α ranges between 0 and 1. When $\alpha \rightarrow 0$, LCM acts in a first-in-first-out manner. When $\alpha \rightarrow 1$, G-EDF/LCM acts like ECM, and G-RMA/LCM acts like RCM. We derived upper bounds on retry cost and response time under LCM. We also compared schedulability of LCM against ECM, RCM and lock-free. We identified the conditions under which LCM performs better than the other synchronization techniques. LCM reduces retry cost of each atmic section to $(1+\alpha_{max})s_{max}$ instead of $2.s_{max}$ in case of ECM and RCM. In ECM and RCM, tasks do not retry due to lower priority tasks, whereas in LCM, they do so. In G-EDF/LCM, retry due to a lower priority job is encountered only from a task τ_j's last job instance during τ_i's period. This is not the case with G-RMA/LCM, because, each higher priority task can be aborted and retried by any job instance of lower priority tasks. Schedulability of G-EDF/LCM and G-RMA/LCM is better or equal to ECM and RCM, respectively, by proper choices for α_{min} and α_{max}. Schedulability of G-EDF/LCM is better than retry-loop lock-free synchronization for G-EDF if the upper bound on s_{max}/r_{max} is between 0.5 and 2, which is higher than that achieved by ECM.

ECM, RCM and LCM suffer from transitive retry in case of multi-objects per transaction. So, we introduced Priority-based with Negative value and First access (PNF) contention manager. PNF avoids transitive retry effect suffered by ECM, RCM and LCM in case of multiple objects per transaction. PNF tries to optimize processor usage by lower priority of aborted transaction. This way, other tasks can proceed if they do no conflict with others. PNF implementation is not as simple as other CMs. For previous contention managers, we upper bounded their retry cost and response times. We compared their schedulability to identify the conditions to prefer one of the them over the others. We also compared their schedulability against schedulability of lock-free method. We also compared retry cost of previous synchronization techniques.

7.2 Post Preliminary-Exam Work

We propose the following post preliminary exam work:

- **Analytical and experimental comparison between developed CMs and real-time locking protocols** It has been said that lock-free and wait-free methods offer numerous advantages over locking protocols, but locking protocols are still of wide use in real-time systems due to simpler programming and analysis than lock-free. Thus, it is desired to compares different CMs against real-time locking protocols. Examples of real-time locking protocols include PCP and its variants [18, 44, 62, 70], multiprocessor PCP (MPCP) [14, 28, 46, 61], SRP [50], multiprocessor SRP (MSRP) [35], PIP [28], FMLP [10,11,43] and OMLP [7]. OMLP and FMLP are similar, and FMLP was found to be superior to other protocols [13].

- **Contention manager development for nested transactions** Transactions can be nested *linearly*, where each transaction has at most one pending transaction [58]. Nesting can also be done in *parallel* where transactions execute concurrently within the same parent [77]. Linear nesting can be *1) flat:* If a child transaction aborts, then parent also aborts. If a child commits, no effect is taken until the parent commits.

Modifications made by child transaction is seen only be the parent . *2) Closed:* Similar to *flat nesting* except that if a child transaction aborts, parent does not have to abort. *3) Open:* If a child transaction commits, its modifications is seen not only by the parent, but also by other non-surrounding transactions. If parent aborts after child commits, child modifications are still valid. It is required to extend the proposed real-time CMs (or develop new ones) to handle some or all types of transaction nesting.

- **Combine both LCM and PNF** LCM is designed to reduce the retry cost of one transaction when it is interfered close to its end of execution. PNF is designed to avoid transitive retry in case of multiple objects per transactions. One goal is to combine benefits of both algorithms.

- **Investigate other criterion for contention managers to further reduced retry cost** Criterion other than or combined with priority, transaction length and first access may be used to produce better contention managers.

Bibliography

[1] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 110–112, New York, NY, USA, 2008. ACM.

[2] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *RTSS*, pages 92–105, 1996.

[3] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *RTSS*, pages 28–37, 1995.

[4] J.H. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications*, pages 57 –64, 2000.

[5] J.H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*, pages 215–234. Kluwer, 1997.

[6] A. Barros and L.M. Pinho. Managing contention of software transactional memory in real-time systems. In *IEEE RTSS, Work-In-Progress*, 2011.

[7] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, pages 119–128, 2007.

[8] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*, pages 149–160, 2007.

[9] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 156–167, New York, NY, USA, 2009. ACM.

[10] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA.

[11] B.B. Brandenburg and J.H. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS-RT. In *RTCSA*, pages 185–194, 2008.

[12] Bjorn B. Brandenburg et al. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS*, pages 342–353, 2008.

[13] Bjrn Brandenburg and James Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmus rt. In Theodore Baker, Alain Bui, and Sbastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 105–124. Springer Berlin / Heidelberg, 2008.

[14] C.M. Chen and S.K. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, UM Computer Science Department, 1998.

[15] H.R. Chen and YH Chin. An efficient real-time scheduler for nested transaction models. In *Proceedings of Ninth International Conference on Parallel and Distributed Systems*, pages 335–340. IEEE, 2002.

[16] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, pages 2 –9, June 1998.

[17] Jing Chen. A loop-free asynchronous data sharing mechanism in multiprocessor real-time systems based on timing properties. In *Proceedings of 23rd International Conference on Distributed Computing Systems Workshops*, pages 184 – 190, May 2003.

[18] M.I. Chen and K.J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[19] H. Cho, B. Ravindran, and E.D. Jensen. Lock-free synchronization for dynamic embedded real-time systems. *ACM Transactions on Embedded Computing Systems*, 9(3):23, 2010.

[20] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373 –384, March 2007.

[21] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. On utility accrual processor scheduling with wait-free synchronization for embedded real-time software. In *Proceedings of the ACM symposium on Applied computing*, SAC '06, pages 918–922, New York, NY, USA, 2006. ACM.

[22] Hyeonjoong Cho, Binoy Ravindran, and E.D. Jensen. A space-optimal wait-free real-time synchronization protocol. In *Proceedings of 17th Euromicro Conference on Real-Time Systems*, ECRTS' 05, pages 79 – 88, July 2005.

[23] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[24] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. Chronos linux: a best-effort real-time multiprocessor linux kernel. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 474–479, New York, NY, USA, 2011. ACM.

[25] UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *ECRTS*, pages 75–84, 2006.

[26] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *International Symposium on Code Generation and Optimization*, CGO '07, pages 21 –33, March 2007.

[27] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[28] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS*, pages 377–386, 2009.

[29] Mohammed El-Shambakey and Binoy Ravindran. On the design of real-time stm contention managers. Technical report, ECE Department, Virginia Tech, 2011. Available as `http://www.real-time.ece.vt.edu/tech-report-rt-stm-cm11.pdf`.

[30] Mohammed Elshambake and Binoy Ravindran. Stm concurrency control for embedded real-time software with tighter time bounds. In *DAC '12, "to appear"*, 2012.

[31] M. Elshambakey and B. Ravindran. Stm concurrency control for multicore embedded real-time software: Time bounds and tradeoffs. In *SAC*, 2012.

[32] S. Fahmy and B. Ravindran. On stm concurrency control for multicore embedded real-time software. In *International Conference on Embedded Computer Systems*, SAMOS, pages 1 –8, July 2011.

[33] S.F. Fahmy, B. Ravindran, and E. D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *DATE*, pages 688–693, 2009.

[34] S.F. Fahmy, B. Ravindran, and ED Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *ACM SAC*, pages 334–338, 2009.

[35] P. Gai, M. Di Natale, et al. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS*, pages 189–198, 2003.

[36] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 144–154. VLDB Endowment, 1981.

[37] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[38] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM.

[39] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2nd. edition, December 2010.

[40] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51:91–100, Aug 2008.

[41] Maurice Herlihy. The art of multiprocessor programming. In *PODC*, pages 1–2, 2006.

[42] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[43] P. Holman and J.H. Anderson. Locking under pfair scheduling. *TOCS*, 24(2):140–174, 2006.

[44] D.K. Kiss. Intelligent priority ceiling protocol for scheduling. In *2011 3rd IEEE International Symposium on Logistics and Industrial Informatics*, LINDI, pages 105 –110, aug. 2011.

[45] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[46] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, pages 469–478, 2009.

[47] K.Y. Lam, T.W. Kuo, and W.H. Tsang. Concurrency control for real-time database systems with mixed transactions. In *Proceedings-Fourth International Workshop on Real-Time Computing Systems and Applications*, pages 96–103. IEEE, 1997.

[48] M.C. Liang, T.W. Kuo, and L.C. Shu. Bap: a class of abort-oriented protocols based on the notion of compatibility. In *Proceedings of Third International Workshop on Real-Time Computing Systems and Applications*, pages 118–127. IEEE, 1996.

[49] M.C. Liang, T.W. Kuo, and L.C. Shu. A quantification of aborting effect for real-time data accesses. *IEEE Transactions on Computers*, 52(5):670–675, 2003.

[50] JM Lopez, JL Diaz, and DF Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[51] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[52] J. Manson, J. Baker, et al. Preemptible atomic regions for real-time Java. In *RTSS*, pages 10–71, 2006.

[53] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, LCR '04, pages 1–7, New York, NY, USA, 2004. ACM.

[54] V.J. Marathe, M.F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W.N. Scherer III, and M.L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT.

[55] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 18–29, New York, NY, USA, 2002. ACM.

[56] Austen McDonald. *Architectures for Transactional Memory.* PhD thesis, Stanford University, June 2009.

[57] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[58] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186 – 201, 2006.

[59] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 184–196, New York, NY, USA, 2002. ACM.

[60] Sathya Peri and Krishnamurthy Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 95–106, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, pages 116–123, 2002.

[62] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[63] Ragunathan Rajkumar, editor. *Real-Time Database Systems: Architecture and Techniques*. Kluwer Academic, 2001.

[64] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 5–17, New York, NY, USA, 2002. ACM.

[65] Bratin Saha, Ali-Reza Adl-Tabatabai, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[66] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[67] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *RTCSA*, pages 477–485, 2009.

[68] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[69] M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *ACM SAC*, pages 326–333, 2010.

[70] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, pages 1175–1185, 1990.

[71] N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.

[72] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 78–88, New York, NY, USA, 2007. ACM.

[73] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 104–115, New York, NY, USA, 2007. ACM.

[74] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 141–150, New York, NY, USA, 2009. ACM.

[75] S.M. Tseng, YH Chin, and W.P. Yang. An adaptive value-based scheduling policy for multiprocessor real-time database systems. In *Proceedings of Eighth International Workshop on Database and Expert Systems Applications*, pages 254–259. IEEE, 1997.

[76] P. Tsigas and Yi Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pages 247 –254, 1999.

[77] H. Volos, A. Welc, A.R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. *ECOOP 2009–Object-Oriented Programming*, pages 123–147, 2009.

[78] W. Xian-De, S. Zhao-Wei, and X. Ze-Jing. A data-centered transaction scheduling strategy of realtime database in micro-satellite ground test system. In *International Conference on Mechatronics and Automation*, ICMA' 09, pages 2952–2956. IEEE, 2009.

[79] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.

[80] Z. Yuehua and Q. Jing. A new multi-dynamic priority real-time database scheduling algorithm. In *2nd International Conference on Computer Engineering and Technology*, volume 7 of *ICCET*, pages V7–177. IEEE, 2010.