Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations

Mohammed El-Shambakey

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

May 30, 2012
Blacksburg, Virginia

Keywords: Software Transactional Memory, Embedded Systems, Contention Managers

Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations

Mohammed El-Shambakey

(ABSTRACT)

Lock-based concurrency control suffers from programmability, scalability, and composability challenges. These challenges are exacerbated in emerging multicore architectures, on which improved software performance must be achieved by exposing greater concurrency. Transactional memory (TM) is an alternative synchronization model for shared memory objects that promises to alleviate these difficulties.

In this dissertation proposal, we consider software transactional memory (STM) for concurrency control in multicore real-time software, and present a suite of real-time STM contention managers for resolving transactional conflicts. The contention managers are called RCM, ECM, LCM, and PNF. RCM and ECM resolve conflicts using fixed and dynamic priorities of real-time tasks, respectively, and are naturally intended to be used with the fixed priority (e.g., G-RMA) and dynamic priority (e.g., G-EDF) multicore real-time schedulers, respectively. LCM resolves conflicts based on task priorities as well as atomic section lengths, and can be used with G-EDF or G-RMA. Transactions under ECM, RCM and LCM can retry due to non-shared objects with higher priority tasks. PNF avoids this problem.

We establish upper bounds on transactional retry costs and task response times under all the contention managers through schedulability analysis. Since ECM and RCM conserve the semantics of the underlying real-time scheduler, their maximum transactional retry cost is double the maximum atomic section length. This is improved in the the design of LCM, which achieves shorter retry costs. However, ECM, RCM, and LCM are affected by transitive retries when transactions access multiple objects. Transitive retry causes a transaction to abort and retry due to another non-conflicting transaction. PNF avoids transitive retry, and also optimizes processor usage by lowering the priority of retrying transactions, enabling other non-conflicting transactions to proceed.

We also formally compare the proposed contention managers with lock-free synchronization. Our comparison reveals that, for most cases, ECM, RCM, G-EDF(G-RMA)/LCM achieve higher schedulability than lock-free synchronization only when the atomic section length does not exceed half of the lock-free retry loop length. Under PNF, atomic section length can equal length of retry loop. With low contention, atomic section length under ECM can equal retry loop length while still achieving better schedulabiltiy. While in RCM, atomic section length can exceed retry loop length. By adjustment of LCM design parameters, atomic section length can be of twice length of retry loop under G-EDF/LCM. While under G-RMA/LCM, atomic section length can exceed length of retry loop.

We implement the contention managers in the Rochester STM framework and conduct experimental studies using a multicore real-time Linux kernel. Our studies confirm that, the contention managers achieve orders of magnitude shorter retry costs than lock-free synchronization. Among the contention managers, PNF performs the best.

Building upon these results, we propose real-time contention managers that allow nested atomic sections – an open problem – for which STM is the only viable non-blocking synchronization solution. Optimizations of LCM and PNF to obtain improved retry costs and greater schedulability advantages are also proposed.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Embedded systems sense physical processes and control their behavior, typically through feedback loops. Since physical processes are concurrent, computations that control them must also be concurrent, enabling them to process multiple streams of sensor input and control multiple actuators, all concurrently. Often, such computations need to concurrently read/write shared data objects. Typically, they must also process sensor input and react, satisfying application-level time constraints.

The de facto standard for programming concurrency is the threads abstraction, and the de facto synchronization abstraction is locks. Lock-based concurrency control has significant programmability, scalability, and composability challenges [58]. Coarse-grained locking (e.g., a single lock guarding a critical section) is simple to use, but permits no concurrency: the single lock forces concurrent threads to execute the critical section sequentially, in a one-at-a-time order. This is a significant limitation, especially with the emergence of multicore architectures, on which improved software performance must be achieved by exposing greater concurrency.

With fine-grained locking, a single critical section is broken down into several critical sections – e.g., each bucket of a hash table is guarded by a unique lock. Thus, threads that need to access different buckets can do so concurrently, permitting greater parallelism. However, this approach has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion.

Perhaps, the most significant limitation of lock-based code is its non-composability. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is not possible in a straightforward manner: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety.

Lock-free synchronization [57], which uses atomic hardware synchronization primitives (e.g., Compare And Swap [67,68], Load-Linked/Store-Conditional [106]), also permits greater concurrency, but has even lower programmability: lock-free algorithms must be custom-designed for each situation (e.g., a data structure [21,48,56,60,87]). Additionally, it is not clear how to program nested critical sections using lock-free synchronization. Most importantly, reasoning about the correctness of lock-free algorithms is significantly difficult [57].

1.1 Transactional Memory

Transactional memory (TM) is an alternative synchronization model for shared memory data objects that promises to alleviate these difficulties. With TM, programmers write concurrent code using threads, but organize code that read/write shared memory objects as *memory transactions*, which speculatively execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. Objects read and written by transactions are also monitored, in read sets and write sets, respectively. Two transactions conflict if they access the same object and one access is a write. (Conflicts are usually detected by detecting non-empty read and write set intersections.) When that happens, a contention manager (CM) resolves the conflict by aborting one and committing the other, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffers (lazy).

In addition to a simple programming model (locks are excluded from the programming interface), TM provides performance comparable to lock-based synchronization [96], especially for high contention and read-dominated workloads, and is composable. TM's first implementation was proposed in hardware, called hardware transactional memory (or HTM) [59]. HTM has the lowest overhead, but HTM transactions are usually limited in space and time. Examples of HTMs include TCC [55], UTM [1], Oklahoma [108], ASF [32], and Bulk [24]. TM implementation in software, called software transactional memory (or STM) was proposed later [104]. STM transactions do not need any special hardware, are not limited in size or time, and are more flexible. However, STM has a higher overhead, and thus lower performance, than HTM. Examples of STMs include RSTM [112], TinySTM [95], Deuce [70], and AtomJava [61].

Listing 1.1: STM example

```
BEGIN_TRANSACTION;
        stm::wr_ptr<Counter> wr(m_counter);
        wr->set_value(wr->get_value(wr) + 1, wr);
END_TRANSACTION;
```

Listing 1.1 shows an example STM code written by RSTM [105]'s interface. RSTM's `BEGIN_TRANSACTION` and `END_TRANSACTION` keywords are used to enclose a critical section,

which creates a transaction for the enclosed code block and guarantees its atomic execution. First line inside the transaction makes a write pointer to a variable "m_counter" of type "Counter". The second line reads the current value of the counter variable through "wr−>get_value". The counter value is incremented through "wr−>set_value" operation.

Hybrid TM (or HyTM) was subsequently proposed in [79], which combines HTM with STM, and avoids their limitations. Examples of HyTMs include SpHT [77], VTM [93], HyTM [33], LogTM [88], and LogTM-SE [114].

1.2 STM for Real-Time Software

Given the hardware-independence of STM, which is a significant advantage, we focus on STM. STM's programmability, scalability, and composability advantages are also compelling for concurrency control in multicore embedded real-time software. However, this will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy application-level time constraints. Transactional retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are OS scheduler-dependent).

Despite the large body of work on STM contention managers, relatively few results are known on real-time contention management. STM concurrency control for real-time systems has been previously studied, but in a limited way. For example, [83] proposes a restricted version of STM for uniprocessors. Uniprocessors do not need contention management. [46] bounds response times in distributed multicore systems with STM synchronization. They consider Pfair scheduling [66], which is largely only of theoretical interest[1], limit to small atomic regions with fixed size, and limit transaction execution to span at most two quanta. [97] presents real-time scheduling of transactions and serializes transactions based on transactional deadlines. However, the work does not bound transactional retries and response times.

[100] proposes real-time HTM, which of course, requires hardware with TM support. The retry bound developed in [100] assumes that the worst case conflict between atomic sections of different tasks occurs when the sections are released at the same time. We show that, this assumption does not cover the worst case scenario (see Chapter 4). [45] presents a contention manager that resolves conflicts using task deadlines. The work also establishes upper bounds on transactional retries and task response times. However, similar to [100], [45] also assumes that the worst case conflict between atomic sections occurs when the sections are released simultaneously. Besides, [45] assumes that all transactions have equal lengths. The ideas in [45] are extended in [9], which presents three real-time CM designs. But no retry bounds or schedulability analysis techniques are presented for those CMs.

[1] This is due to Pfair class of algorithm's time quantum-driven nature of scheduling and consequent high run-time overheads.

Thus, past efforts on real-time STM are limited, and do not answer important fundamental questions:

(1) How to design "general purpose" real-time STM contention managers for multicore architectures? By general purpose, we mean those that do not impose any restrictions on transactional properties (e.g., transaction lengths, number of transactional objects, levels of transactional nestings), which are key limitations of past work.

(2) What tight upper bounds exist for transactional retries and task response times under such real-time CMs?

(3) How does the schedulability of real-time CMs compare with that of lock-free synchronization? i.e., are there upper bounds or lower bounds for transaction lengths below or above which is STM superior to lock-free synchronization?

(4) How does transactional retry costs and task response times of real-time CMs compare with that of lock-free synchronization in practice (i.e., on average)?

1.3 Research Contributions

In this dissertation proposal, we answer these questions. We present a suite of real-time STM contention managers, called RCM, ECM, LCM, and PNF. The contention managers progressively improve transactional retry and task response time upper bounds (and consequently improve STM's schedulability advantages) and also relax the underlying task models. RCM and ECM resolve conflicts using fixed and dynamic priorities of real-time tasks, respectively, and are naturally intended to be used with the fixed priority (e.g., G-RMA [22]) and dynamic priority (e.g., G-EDF [22]) multicore real-time schedulers, respectively. LCM resolves conflicts based on task priorities as well as atomic section lengths, and can be used with G-EDF or G-RMA. Transactions under ECM, RCM and LCM can restart because of other transactions that share no objects with them. This is called transitive retry. PNF solves this problem. PNF also optimizes processor usage through reducing priority of aborted transactions. So, other tasks can proceed.

We establish upper bounds on transactional retry costs and task response times under all the contention managers through schedulability analysis. Since ECM and RCM conserve the semantics of the underlying real-time scheduler, their maximum transactional retry cost is double the maximum atomic section length. This is improved in the design of LCM, which achieves shorter retry costs. However, ECM, RCM, and LCM are affected by transitive retries when transactions access multiple objects. Transitive retry causes a transaction to abort and retry due to another non-conflicting transaction. PNG avoids transitive retry, and also optimizes processor usage by lowering the priority of retrying transactions, enabling other non-conflicting transactions to proceed.

We formally compare the schedulability of the proposed contention managers with lock-free synchronization. Our comparison reveals that, for most cases, ECM and RCM achieve higher schedulability than lock-free synchronization only when the atomic section length does not exceed half of lock-free synchronization's retry loop length. However, in some cases, the atomic section length can reach the lock-free retry loop length for ECM and it can even be larger than the lock-free retry loop-length for RCM, and yet higher schedulability can be achieved with STM. This means that, STM is more advantageous with G-RMA than with G-EDF.

LCM achieves shorter retry costs and response times than ECM and RCM. Importantly, the atomic section length range for which STM's schedulability advantage holds is significantly expanded with LCM (over that under ECM and RCM): Under ECM, RCM and LCM, transactional length should not exceed half of lock-free retry loop length to achieve better schedulability. However, with low contention, transactional length can increase to retry loop length under ECM. Under RCM, transactional length can be of many orders of magnitude of retry loop length with low contention. With suitable LCM parameters, transactional length under G-EDF/LCM can be twice as retry loop length. While in G-RMA/LCM, transactional length can be of many orders of magnitude as retry loop length. PNF achieves better schedulability than lock-free as long as transactional length does not exceed length of retry loop.

Why are we concerned about expanding STM's schedulability advantage? When STM's schedulability advantage holds, programmers can reap STM's significant programmability and composability benefits in multicore real-time software. Thus, by expanding STM's schedulability advantage, we increase the range of real-time software for which those benefits can be tapped. Our results, for the first time, thus provides a fundamental understanding of when to use, and not use, STM concurrency control in multicore real-time software.

We also implement the contention managers in the RSTM framework [105] and conduct experimental studies using the ChronOS multicore real-time Linux kernel [35]. Our studies confirm that, the contention managers achieve shorter retry costs than lock-free synchronization by as much as 95% improvement (on average). Among the contention managers, PNF performs the best in case of high transitive retry. PNF achieves shorter retry costs than ECM, RCM and LCM by as much as 53% improvement (on average).

1.4 Summary of Proposed Post Preliminary Research

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

Supporting nested transactions. Transactions can be nested *linearly*, where each transaction has at most one pending transaction [89]. Nesting can also be done in *parallel* where transactions execute concurrently within the same parent [113]. Linear nesting can be *1) flat:* If a child transaction aborts, then the parent transaction also aborts. If a child commits, no

effect is taken until the parent commits. Modifications made by the child transaction are only visible to the parent until the parent commits, after which they are externally visible. *2) Closed:* Similar to *flat nesting*, except that if a child transaction conflicts, it is aborted and retried, without aborting the parent, potentially improving concurrency over flat nesting. *3) Open:* If a child transaction commits, its modifications are immediately externally visible, releasing memory isolation of objects used by the child, thereby potentially improving concurrency over closed nesting. However, if the parent conflicts after the child commits, then compensating actions are executed to undo the actions of the child, before retrying the parent and the child. We propose to develop real-time contention managers that allow these different nesting models and establish their retry and response time upper bounds. Additionally, we propose to formally compare their schedulability with nested critical sections under lock-based synchronization. Note that, nesting is not viable under lock-free synchronization.

Combinations and optimizations of LCM and PNF contention managers. LCM is designed to reduce the retry cost of a transaction when it is interfered close to the end of its execution. In contrast, PNF is designed to avoid transitive retry when transactions access multiple objects. An interesting direction is to combine the two contention managers to obtain the benefits of both algorithms. Further design optimizations may also be possible to reduce retry costs and response times, by considering additional criteria for resolving transactional conflicts. Importantly, we must also understand what are the schedulability advantages of such a combined/optimized CM over that of LCM and PNF, and how such a combined/optimized CM behaves in practice. This will be our second research direction.

Formal and experimental comparison with real-time locking protocols. Lock-free synchronization offers numerous advantages over locking protocols, but (coarse-grain) locking protocols have had significant traction in real-time systems due to their good programmability (even though their concurrency is low). Example such real-time locking protocols include PCP and its variants [26, 69, 92, 102], multicore PCP (MPCP) [73, 91], SRP [8, 23], multicore SRP (MSRP) [49], PIP [38], FMLP [16, 17, 64], and OMLP [11]. OMLP and FMLP are similar, and FMLP has been established to be superior to other protocols [20]. How does their schedulability compare with that of the proposed contention managers? How do they compare in practice? These questions constitute our third research direction.

1.5 Proposal Organization

The rest of this dissertation proposal is organized as follows. Chapter 2 overviews past and related work on real-time concurrency control. Chapter 3 describes our task/system model and assumptions. Chapter 4 describes the ECM and RCM contention managers, derives upper bounds for their retry costs and response times, and compares their schedulability between themselves and with lock-free synchronization. Chapters 5 and 6 similarly describe the LCM and PNF contention managers, respectively. Chapter 7 describes our implementation and reports our experimental studies. We conclude in Chapter 8.

Chapter 2

Past and Related Work

Many mechanisms appeared for concurrency control for real-time systems. These methods include locking [23, 78], lock-free [3–5, 27, 30, 36, 43, 44, 65, 72] and wait-free [2, 12, 25, 27, 28, 31, 44, 62, 94, 109–111]. In general, real-time locking protocols have disadvantages like: 1) serialized access to shared object, resulting in reduced concurrency and reduced utilization. 2) increased overhead due to context switches. 3) possibility of deadlock when lock holder crashes. 3) some protocols requires apriori knowledge of ceiling priorities of locks. This is not always available. 4) Operating system data structures must be updates with this knowledge which reduces flexibility. For real-time lock-free, the most important problem is to bound number of failed retries and reduce cost of a single loop. The general technique to access lock-free objects is "retry-loop". Retry-loop uses atomic primitives (e.g., CAS) which is repeated until success. To access a specific data structure efficiently, lock-free technique is customized to that data structure. This increases difficulty of response time analysis. Primitive operations do not access multiple objects concurrently. Although some attempts made to enable multi-word CAS [3], but it is not available in commodity hardware [86]. For real-time wait-free protocols. It has a space problem due to use of multiple buffers. This is inefficient in some applications like small-memory real-time embedded systems. Wait-free has the same problem of lock-free in handling multiple objects.

The rest of this Chapter is organized as follows, Section 2.1 summarizes previous work on real-time locking protocols. In Section 2.2, we preview related work on lock-free and wait-free methods for real-time systems. Section 2.3 provides concurrency control under real-time database systems as a predecessor and inspiration for real-time STM. Section 2.4 previews related work on contention management. Contention management policy affects response time analysis of real-time STM.

2.1 Real-Time Locking Protocols

A lot of work has been done on real-time locking protocols. Locks in real-time systems can lead to priority inversion [23, 78]. Under priority inversion, a higher priority job is not allowed to run because it needs a resource locked by a lower priority job. Meanwhile, an intermediate priority job preempts the lower priority one and runs. Thus, the higher priority job is blocked because of a lower priority one. Different locking protocols appeared to solve this problem, but exposing other problems. Most of real-time blocking protocols are based on *Priority Inheritance Protocol (PIP)* [23, 38, 102], *Priority Ceiling Protocol (PCP)* [23, 26, 38, 69, 73, 91, 92, 102] and *Stack Resource Protocol (SRP)* [8, 23, 50].

In PIP [23, 102], resource access is done in FIFO order. A resource holder inherits highest priority of jobs blocked on that resource. When resource holder releases the resource and it holds no other resources, its priority is returned to its normal priority. If it holds other resources, its priority is returned to highest priority job blocked on other resources. Under PIP, a high priority job can be blocked by lower priority jobs for at most the minimum of number of lower priority jobs and number of shared resources. PIP suffers from chained blocking, in which a higher priority task is blocked for each accessed resource. Besides, PIP suffers from deadlock where each of two jobs needs resources held by the other. So, each job is blocked because of the other. [38] provides response time analysis for PIP when used with fixed-priority preemptive scheduling on multiprocessor system.

PCP [23, 92, 102] provides concept of priority ceiling. Priority ceiling of a resource is the highest priority of any job that can access that resource. For any job to enter a critical section, its priority should be higher the priority ceiling of any currently accessed resource. Otherwise, the resource holder inherits the highest priority of any blocked job. Under PCP, a job can be blocked for at most one critical section. PCP prevents deadlocks. [26] extends PCP to dynamically scheduled systems.

Two protocols extend PCP to multiprocessor systems: 1) *Multiprcoessor PCP (M-PCP)* [73, 91, 92] discriminates between global resources and local resources. Local resources are accessed by PCP. A global resource has a base priority greater than any task normal priority. Priority ceiling of a global resource equals sum of its base priority and highest priority of any job that can access it. A job uses a global resource at the priority ceiling of that resource. Requests for global resources are enqueued in a priority queue according to normal priority of requesting job. 2) *Parallel-PCP (P-PCP)* [38] extends PCP to deal with fixed priority preemptive multiprocessor scheduling. P-PCP, in contrast to PCP, allows lower priority jobs to allocate resources when higher priority jobs already access resources. Thus, increasing parallelism. Under P-PCP, a higher priority job can be blocked multiple times by a lower priority job. With reasonable priority assignment, blocking time by lower priority jobs is small. P-PCP uses α_i parameter to specify permitted number of jobs with basic priority lower than i and effective priority higher than i. When α_i is small, parallelism is reduced, so as well blocking from lower priority tasks. Reverse is true. [38] provides response time

analysis for P-PCP.

[80] extends P-PCP to provide *Limited-Blocking PCP (LB-PCP)*. LB-PCP provides more control on indirect blocking from lower priority tasks. LB-PCP specify additional counters that control number of times higher priority jobs can be indirectly blocked without the need of reasonable priority assignment as in P-PCP. [80] analyzes response time of LB-PCP and experimentally compares it to P-PCP. Results show that LB-PCP is appropriate for task sets with medium utilization.

PCP can be unfair from blocking point of view. PCP can cause unnecessary and long blocking for tasks that do not need any resources. Thus, [69] provides Intelligent PCP (IPCP) to increase fairness and to work in dynamically configured system (i.e., no a priori information about number of tasks, priorities and accessed resources). IPCP initially optimizes priorities of tasks and resources through learning. Then, IPCP tunes priorities according to system wide parameters to achieve fairness. During the tuning phase, penalties are assigned to tasks according to number of higher priority tasks that can be blocked.

SRP [8, 23, 50] extends PCP to allow multiunit resources and dynamic priority scheduling and sharing runtime stack-based resources. SRP uses *preemption level* as a static parameter assigned to each task despite its dynamic priority. Resource ceiling is modified to include number of available resources and preemption levels. System ceiling is the highest resource ceiling. A task is not allowed to preempt unless it is the highest priority ready one, and its preemption level is higher than the system ceiling. Under SRP, a job can be blocked at most for one critical section. SRP prevents deadlocks. *Multiprocessor SRT (M-SRP)* [49] extends SRP to multiprocessor systems. M-SRP, as M-PCP, discriminates between local and global resources. Local resources are accessed by SRP. Request for global resource are enqueued in a FIFO queue for that resource. Tasks with pending requests busy-wait until their requests are granted.

Another set of protocols appeared for PFair scheduling [16]. [63] provide initial attempts to synchronize tasks with short and long resources under PFair. In Pfair scheduling, each task receives a weight that corresponds to its share in system resources. Tasks are scheduled in quanta, where each quantum has a specific job on a specific processor. Each lock has a FIFO queue. Requesting tasks are ordered in this FIFO queue. If a task is preempted during critical section, then other tasks can be blocked for additional time known as *frozen time*. Critical sections requesting short resources execute at most in two quanta. By early lock-request, critical section can finish in one quanta, avoiding the additional blocking time. [63] proposes two protocols to deal with short resources: 1) *Skip Protocol (SP)* leaves any lock request in the FIFO queue during frozen interval until requesting task is scheduled again. 2) *Rollback Protocol (RP)* discards any request in the FIFO queue for the lock during frozen time. For long resources, [63] uses *Static Weight Server Protocol (SWSP)* where requests for each resource l is issued to a corresponding server S. S orders requests in a FIFO queue and has a static specific weight.

Flexible Multiprocessor Locking Protocol (FMLP) [16] is the most famous synchronization

protocol for PFair scheduling. The FMLP allows non-nested and nested resources access without constraints. FMLP is used under global and partitioned deadline scheduling. Short or long resource is user defined. Resources can be grouped if they are nested by some task and have the same type. Request to a specific resource is issued to its containing group. Short groups are protected by non-preemptive FIFO queue locks, while long groups are protected by FIFO semaphore queues. Tasks busy-wait for short resources and suspend on long resources. Short request execute non-preemptively. Requests for long resources cannot be contained within requests for short resources. A job executing a long request inherits highest priority of blocked jobs on that resource's group. FMLP is deadlock free.

[18] is concerned with suspension protocols. Schedulability analysis for suspension protocols can be suspension-oblivious or suspension-aware. In suspension-oblivious, suspension time is added to task execution. While in suspension-aware, it is not. [18] provides *Optimal Multiprocessor Locking Protocol (OMLP)*. Under OMLP, each resource has a FIFO queue of length at most m, and a priority queue. Requests for each resource are enqueued in the corresponding FIFO queue. If FIFO queue is full, requests are added to the priority queue according to the requesting job's priority. The head of the FIFO queue is the resource holding task. Other queued requests are suspended until their turn come. OMLP achieves $O(m)$ priority inversion (pi) blocking per job under suspension oblivious analysis. This is why OMLP is asymptotically optimal under suspension oblivious analysis. Under suspension aware analysis, FMLP is asymptotically optimal. [19] extends work in [18] to clustered-based scheduled multiprocessor system. [19] provides concept of *priority donation* to ensure that each job is preempted at most once. In priority donation, a resource holder priority can be unconditionally increased. Thus, a resource holder can preempt another task. The preempted task is predetermined such that each job is preempted at most once. OMLP with priority donation can be integrated with k-exclusion locks (K-OMLP). Under K-exclusion locks, there are k instances of the same resource than can be allocated concurrently. K-OMLP has the same structure of OMLP except that there are K FIFO queues for each resource. Each FIFO queue corresponds to one of the k instances. K-OMLP has $O(m/k)$ bound for pi-blocking under s-oblivious analysis. [39] extends the K-OMLP in [19] to global scheduled multiprocessor systems. The new protocol is *Optimal K-Exclusion Global Locking Protocol (O-KGLP)*. Despite global scheduling is a special case of clustering, K-OMLP provides additional cost to tasks requesting no resources if K-OMLP is used with global scheduling. O-KGLP avoids this problem.

2.2 Real-Time Lock-Free and Wait-Free Synchronization

Due to locking problems (e,g,. priority inversion, high overhead and deadlock), research has been done on non-blocking synchronization using lock-free [3–5,27,30,43,44,65,72] and wait-free algorithms [2,12,25,27,28,31,44,62,94,109–111]. Lock-free iterates an atomic primitive

(e.g., CAS) inside a retry loop until successfully accessing object. When used with real-time systems, number of failed retries must be bounded [3, 4]. Otherwise, tasks are highly likely to miss their deadlines. Wait-free algorithms, on the other hand, bound number of object access by any operation due to use of sized buffers. Synchronization under wait-free is concerned with: 1) single-writer/multi-readers where a number of reading operations may conflict with one writer. 2) multi-writer/multi-reader where a number of reading operations may conflict with number of writers. The problem with wait-free algorithms is its space cost. As embedded real-time systems are concerned with both time and space complexity, some work appeared trying to combine benefits of locking and wait-free.

[4] considers lock-free synchronization for hard-real time, periodic, uniprocessor systems. [4] upper bounds retry loop failures and derives schedulability conditions with Rate Monotonic (RM), and Earliest Deadline First (EDF). [4] compares, formally and experimentally, lock-free objects against locking protocols. [4] concludes that lock-free objects often require less overhead than locking-protocols. They require no information about tasks and allow addition of new tasks simply. Besides, lock-free object do not induce excessive context switches nor priority inversion. On the other hand, locking protocols allow nesting. Besides, performance of lock-free depends on the cost of "retry-loops". [3] extends [4] to generate a general framework for implementing lock-free objects in uniprcocessr real-time systems. The framework tackles the problem of multi-objects lock-free operations and transactions through multi-word compare and swap (MWCAS) implementation. [3] provides a general approach to calculate cost of operation interference based on linear programming. [3] compares the proposed framework with real-time locking protocols. Lock-free objects are prefered if cost of retry-loop is less than cost of lock-access-unlock sequence. [5] extends [3, 4] to use lock-free objects in building memory-resident transactions for uniprocessor real-time systems. Lock-free transactions, in contrast to lock-based transactions, do not suffer from priority inversion, deadlocks, complicated data-logging and rolling back. Lock-free transaction do not require kernel support.

[36] presents two synchronization methods under G-EDF scheduled real-time multiprocessor systems for simple objects. The first synchronization technique uses queue-based spin locks, while the other uses lock-free. The queue lock is FIFO ordered. Each task appends an entry at the end of the queue, and spins on it. While the task is spinning, it is non-preemptive. The queue could have been priority-based but this complicates design and does not enhance worst case response time analysis. Spinning is suitable for short critical sections. Disabling preemption requires kernel support. So, second synchronization method uses lock-free objects. [36] bounds number of retries. [36] , analytically and experimentally, evaluates both synchronization techniques for soft and hard real-time analysis. [36] concludes that queue locks have a little overhead. They are suitable for small number of shared object operations per task. Queue locks are not generally suitable for nesting. Lock-free have high overhead compared with queue locks. Lock-free is suitable for small number of processors and object calls in the absence of kernel support.

[65] uses lock-free objects under PFair scheduling for multiprocessor system. [65] provides

concept of *supertasking* to reduce contention and number of failed retries. This is done by collecting jobs that need a common resource into the same supertask. Members of the same supertask run on the same processor. Thus, they cannot content together. [65] upper bounds worst case duration for lock-free object access with and without supertasking. [65] optimizes, not replaces, locks by lock-free objects. Locks are still used in situations like sharing external devices and accessing complex objects.

Lock-free objects are used with time utility models where importance and criticality of tasks are separated [30,72]. [72] presents *MK-Lock-Free Utility Accrual (MK-LFUA)* algorithm that minimizes system level energy consumption with lock-free synchronization. [30] uses lock-free synchronization for dynamic embedded real-time systems with resource overloads and arbitrary activity arrivals. Arbitrary activity arrivals are modelled with Universal Arrival Model (UAM). Lock-free retries are upper bounded. [30] identifies the conditions under which lock-free is better than lock-based sharing. [43] builds a lock-free linked-list queue on a multi-core ARM processor.

Wait-free protocols use multiple buffers for readers and writers. For single-writer/multiple-readers, each object has a number of buffers proportional to maximum number of reader's preemptions by the writer. This bounds number of reader's preemptions. Readers and writers can use different buffers without interfering each other.

[31] presents wait-free protocol for single-writer/multiple-readers in small memory embedded real-time systems. [31] proves space optimality of the proposed protocol, as it required the minimum number of buffers. The protocol is safe and orderly. [31] also proves, analytically and experimentally, that the protocol requires less space than other wait-free protocols. [28] extends [31] to present wait-free utility accrual real-time scheduling algorithms (RUA and DASA) for real-time embedded systems. [28] derives lower bounds on accrued utility compared with lock-based counterparts while minimizing additional space cost. Wait-free algorithms experimentally exhibit optimal utility for step time utility functions during underload, and higher utility than locks for non-step utility functions. [94] uses wait-free to build three types of concurrent objects for real-time systems. Built objects has persistent states even if they crash. [111] provides wait-free queue implementation for real-time Java specifications.

A number of wait-free protocols were developed to solve multi-writer/multi-reader problem in real-time systems. [110] provides m-writer/n-reader non-blocking synchronization protocol for real-time multiprocessor system. The protocol needs $n + m + 1$ slots. [110] provides schedulability analysis of the protocol. [2] presents wait-free methods for multi-writer/multi-reader in real-time multiprocessor system. The proposed algorithms are used for both priority and quantum based scheduling. For a B word buffer, the proposed algorithms exhibit $O(B)$ time complexity for reading and writing, and $\Theta(B)$ space complexity. [109] provides a space-efficient wait-free implementation for n-writer/n-reader synchronization in real-time multiprocessor system. The proposed algorithm uses timestamps to implement the shared buffer. [109] uses real-time properties to bound timestamps. [25] presents wait-free implementation of the multi-writer/multi-reader problem for real-time multiprocessor

synchronization. The proposed mechanism replicates single-writer/multi-reader to solve the multi-writer/multi-reader problem. [25], as [109], uses real-time properties to ensure data coherence through timestamps.

Each synchronization technique has its benefits. So, a lot of work compares between locking, lock-free and wait-free algorithms. [44] compares building snapshot tool for real-time system using locking, lock-free and wait-free. [44] analytically and experimentally compares the three methods. [44] concludes that wait-free is better than its competitors. [27] presents synchronization techniques under LNREF [29] (an optimal real-time multiprocessor scheduler) for simple data structures. Synchronization mechanisms include lock-based, lock-free and wait-free. [27] derives minimum space cost for wait-free synchronization. [27] compares, analytically and experimentally, between lock-free and lock-based synchronization under LNREF.

Some work tried to combine different synchronization techniques to combine their benefits. [62] uses combination of lock-free and wait-free to build real-time systems. Lock-free is used only when CAS suffices. The proposed design aims at allowing good real-time properties of the system, thus better schedulability. The design also aims at reducing synchronization overhead on uni and multiprocessor systems. The proposed mechanism is used to implement a micro-kernel interface for a uni-processor system. [12] combines locking and wait-free for real-time multiprocessor synchronization. This combination aims to reduce required space cost compared to pure wait-free algorithms, and blocking time compared to pure locking algorithms. The proposed scheme is jsut an idea. No formal analysis nor implementation is provided.

2.3 Real-Time Database Concurrency Control

Real-time database systems (RTDBS) is not a synchronization technique. It is a predecessor and inspiration for real-time transactional memory. RTDBS itself uses synchronization techniques when transactions conflict together. RTDBS is concerned not only with logical data consistency, but also with temporal time constraints imposed on transactions. Temporal time constraints require transactions finish before their deadlines. External constraints require updating temporal data periodically to keep freshness of database. RTDBS allow mixed types of transactions. But a whole transaction is of one type. In real-time TM, a single task may contain atomic and non-atomic sections.

High-Priority two Phase Locking (HP-2PL) protocol [74, 75, 90, 116] and *Real-Time Optimistic Concurrency (RT-OCC)* protocol [34, 47, 74–76, 116] are the most two common protocols for RTDBS concurrency . HP-2PL works like 2PL except that when a higher priority transaction request a lock held by a lower priority transaction, lower priority transaction releases the lock in favor of the higher priority one. Then, lower priority transaction restarts. RT-OCC delays conflict resolution till transaction validation. If validating transaction cannot

be serialized with conflicting transactions, a priority scheme is used to determine which transaction to restart. In *Optimistic Concurrency Control with Broadcast Commit (OCC-BC)*, all conflicting transactions with the validating one are restarted. HP-2PL may encounter deadlock and long blocking times, while transactions under RT-OCC suffer from restart time at validation point.

Other protocols were developed based on HP-2PL [74, 75, 90] and RT-OCC [7, 47, 74, 76]. HP-2PL, and its derivatives, are similar to locking protocols in real-time systems. They have the same problems in real-time locking protocols like priority inversion. So, the same solutions exist for the RTDBS locking protocols. Despite RT-OCC, and its derivatives, use locks in their implementation, their behaviour is closer to abort and retry semantics in TM. Some work integrates different protocols to handle different situations [90, 115].

[74] presents *Reduced Ceiling Protocol (RCP)* which is a combination of *Priority Ceiling Protocol (PCP)* and *Optimistic Concurrency Protocol (OCC)*. RCP targets database systems with mixed hard and soft real-time transactions (RTDBS). RCP aims at guarantee of schedulabiltiy of hard real-time transactions, and minimizing deadline miss of soft real-time transactions. Soft real-time transactions are blocked in favor of conflicting hard real-time transactions. While hard real-time transactions use PCP to synchrnonize among themselves, soft real-time transactions use OCC. Hard real-time transactions access locks in a *tow phase locking (2PL)* fashion. Seized locks are released as soon as hard real-time transaction no longer need them. This reduces blocking time of soft real-time transactions. [74] derives analytical and experimental evaluation of RCP against other synchronization protocols.

[115], like [74], deals with mixed transaction. [115] classifies mixed transactions into hard (HRT), soft (SRT) and non (NRT) real-time transactions. HRT has higher priority than SRT. SRT has higher priority than NRT. [115] aims at guranting deadlines of HRTs, minimizing miss rate of SRTs and reducing response time of NRTs. So, [115] deals with inter and intra-transaction concurrency. HRTs use PCP for concurrency control among themselves. SRTs use WAIT-50, and NRTs use 2PL. SRT and NRT are blocked or aborted in favor of HRT. If NRT requests a lock held by SRT, then NRT is blocked. If SRT requests a lock held by NRT, WAIT-50 is applied. Experimental evaluation showed effective improvement in overall system performance. Performance objectives of each transaction type was met.

[47] is concerned with semantic lock concurrency control. The semantic lock technique allows negotiation between logical and temporal constraints of data and transactions. It also controls imprecision resulting from negotiation. Thus, the semantic lock considers scheduling and concurrency of transactions. Semantic lock uses a compatibility function to determine if the release transaction is allowed to proceed or not.

Time Interval OCC protocols try to reduce number of transaction restarts by dynamic adjustment of serialization timestamps. Time interval OCC may encounter unnecessary restarts. [7] presents Timestamp Vector based OCC to resolve these unnecessary restarts. Timestamp Vector base OCC uses a timestamp vector instead of a single timestamp as in Time Interval OCC protocols. Experimental comparison between Timestamp Vector OCC and previous

Time Interval OCC shows higher performance of Timestamp Vector OCC.

[34] aims to investigate performance improvement of priority congnizant OCC over incognizant counterparts. In OCC-BC, all conflicting transactions with the validating transaction are restarted. [34] wonders if it is really worthy to sacrifice all other transactions in favor of one transaction. [34] proposes *Optimistic Concurrency Control- Adaptive PRiority (OCC-APR)* to answer this question. A validating transaction is restarted if it has sufficient time to its deadline if restarted, and higher priority transactions cannot be serialized with the conflicting transaction. Sufficient time estimate is adapted according to system feedback. System feedback is affected by contention level. [34] experimentally concludes that integrating priority into concurrency control management is not very useful. Time Interval OCC showed better performance.

WAIT-X [34, 76] is one of the optimistic concurrency control (OCC) protocols. WAIT-X is a prospective (forward validation) OCC. Prospective means it detects conflicts between a validating transaction and conflicting transaction that may commit in the future. In retrospective (backward validation) protocols, conflicts are detected between a validating transaction and already committed transactions. Retrospective validation aborts validating transaction if it cannot be serialized with already committed conflicting transactions. When WAIT-X detects a conflict, it can either abort validating transaction, or commit validating transaction and abort other conflicting transactions, or it can dealy validating a transction slightly hoping that conflicts resolve themsleves someway. Which action to take is a function of priorities of vlaidating and conflicting transactions. WAIT-X can delay validating transaction until percetage of higher priority transactions in the conflict set is lower than X%. WAIT-50 is a common implementation of WAIT-X.

[71] is concerned with concurrency control for multiprocessor RTDBS. [71] uses priority cap to modify *Reader/Write Prirority Ceiling Protocol (RWPCP)* [103] to work on multiprocessor systems. The proposed protocol, named *One Priority Inversion RWPCP (1PI-RWPCP)*, is deadlock-free and bounds number of priority inversions for any transaction to one. [71] derives feasiblity condition for any transaction under 1PI-RWPCP. [71] experimentally compares performance of 1PI-RWPCP against RWPCP.

[90] combines locking, multi-version and valid confirmation concurrency control mechanisms. The proposed method adopts different concurrency control mechanism according to idiographic situation. Experiments show lower rate of transactional restart of the proposed mechanis compared to 2PL-HP.

[75] is concerned with RTDBS containing periodically updated data and one time transactactions. [75] provides two new concurrency control protocols to balance freshness of data and transaction performance. [75] proposes *HP-2PL with Delayed Restart (HP-2PL-DR)* and *HP-2PL with Delayed Restart and Pre-declaration (HP-2PL-DRP)* based on HP-2PL. Before a transaction T restarts in HP-2PL-DR, next update time of each temporal data accessed by T is checked. If next update time starts before currently re-executing T, then T's restart time is delayed until the next udpate. Otherwise, T is restarted immediately. If T_r and

T_n are two transactions under HP-2PL-DRT. T_r is requesting a lock held by T_n. If priority of T_r is greater than priority of T_n, then T_n releases the lock in favor of T_r. Othewise, T_r fails. If T_n releases the lock and T_n is a one time transaction, then T_n restarts immediately. Otherwise, T_n lock waiting time is updated. Experiments show improved performance of HP-2PL-DR and HP-2PL-DRP over HP-2PL.

2.4 Real-Time TM Concurrency Control

Concurrency control in TM is done through contention managers. Contention managers are used to ensure progress of transactions. If one or more transactions conflict on an object, contention manager decides which transaction to commit. Other transactions abort or wait. Mostly, contention managers are *distributed* or *decentralized* [53, 54, 98, 99], in the sense that each transaction maintains its own contention manager. Contention managers may not know which objects will be needed by transactions and their duration. Past work on contention managers can be classified into two classes: 1) Contention management policy that decides which transaction commits and which do other actions [52–54, 98, 99, 107]. 2) Implementation of contention management policy in practice [15, 37, 51, 82, 98, 107]. The two classes are orthogonal. The second class tries to increase the benefit of the the contention management policy in reality by considering different aspects in TM design (e.g., lazy versus eager, visible versus invisible readers). Second class suggests contention managers should be proactive instead of reactive. This can prevent conflicts before they happen. Contention managers can be supported a lot if they are integrated into system schedulers. This provides a global view of the system (due to applications feedback) and reduces overhead of the implementation of contention manager.

Contention management policy ranges from never aborting enemies to always aborting them [98, 99]. These two extremes can lead to deadlock, starvation, livelock and major loss of performance. Contention manager policy lies in between. Depending on heuristics, contention manager balances between decisions complexity against quality and overhead.

Different types of contention management policies can be found in [52–54, 98, 99, 107] like:

1. Passive and Aggressive: Passive contention manager aborts current transaction, while aggressive aborts enemy.

2. Polite: When conflicting on an object, a transaction spins exponensially for average of $2^{(n+k)}\,ns$, where n is number of times to access the object, and k is a tuning parameter. Spinning times is bounded by m. Afterwards, any enemy is aborted.

3. Karma: It assigns priorities to transaction based on the amount of work done so far. Amount of work is measured by number of opened objects by current transaction. Higher priority transaction aborts lower priority one. If lower priority transaction tries

to access an object for a number of times greater than priority difference between itself and higher priority transaction, enemy is aborted.

4. Eruption: It works like Karma except it adds priority of blocked transaction to the transaction blocking it. This way, enemy is sped-up, allowing blocked transactions to complete faster.

5. Kindergarten: A transaction maintains a hit list (initially empty) of enemies who previously caused current thread to abort. When a new enemy is encountered, current transaction backs off for a limited amount of time. The new enemy is recorded in the hit list. If the enemy is already in the hit list, it is aborted. If current transaction is still blocked afterwards, then it is aborted.

6. Timestamp: It is a fair contention manager. Each transaction gets a timestamp when it begins. Transaction with newer timestamp is aborted in favour of the older. Otherwise, transaction waits for a fixed intervals, marking the enemy flag as defunct. If the enemy is not done afterwards, it is killed. Active transaction clear their flag when they notice it is set.

7. Greedy: Each transaction is given a timestamp when it starts. The earlier the timestamp of a transaction, the higher its priority. If transaction A conflicts with transaction B, and B is of lower priority or is waiting for another transaction, then A aborts B. Otherwise, A waits for B to commit, abort or starts waiting.

8. Randomized: It aborts current transaction with some probability p, and waits with probability $1 - p$.

9. PublishedTimestamp: It works like Timestamp contention manager except it has a new definition for an "inactive" transaction. Each transaction maintains a "recency" flag. Recency flag is updated every time the transaction makes a request. Each transaction maintains its own "inactivity" threshold parameter that is doubled every time it is aborted up to a specific limit. If the enemy "recency" flag is behind the system global time by amount exceeding its "inactivity" threshold, then enemy is aborted.

10. Polka: It is a combination of Polite and Karma contention managers. Like Karma, it assigns priorities based on amount of job done so far. A transaction backs off for a number of intervals equals difference in priority between itself and its enemy. Unlike Karma, back-off length increases exponentially.

11. Prioritized version of some of the previous contention managers appeared. Prioritized contention managers include base priority of the thread holding the transaction into contention manager policy. This way, higher priority threads are more favoured.

[6] compares performance of different contention managers against an optimal, clairvoyant contention manager. The optimal contention manager knows all resources needed by each

transaction, as well as its release time and duration. Comparison is based on the "makespan" concept which is amount of time needed to finish a specific set of transactions. The ratio between makespan of analyzed contention manager and the makespan of the optimal contention manager is known as competitive ratio. [6] proves that any contention manager can be of $O(s)$ competitive ratio if the contention manager is work conserving (i.e., always lets the maximal set of non-conflicting transactions run), and satisfies pending property [53]. The paper proves that this result is asymptotically tight as no on-line work conserving contention manager can achieve better result. [6] also proves that the makespan of greedy contention manager is $O(s)$ instead of $O(s^2)$ [53]. This allows transactions of arbitrary release time and durations in contrast to what is assumed in [53]. For randomized contention managers, a lower bound of $\Omega(s)$ if transaction can modify their resource needs when they are reinvoked.

[52] analyzes different contention managers under different situations. [52] concludes that no single contention manager is suitable for all cases. Thus, [52] proposes a polymorphic contention manager that changes contention managers on the fly throughout different loads, concurrent threads of single load and even different phases of a single thread. To implement polymorphic contention manager, it is important to resolve conflicts resulting from different contention managers in the same application by different methods. The easiest way is to abort the enemy contention manager if it is of different type. [52] uses generic priorities for each transaction regardless of the transaction's contention manager. Upon conflict between different classes of contention manager, highest priority transaction is committed.

[107] provides a comprehensive contention manager attempting to achieve low overhead for low contention, and good throughput and fairness in case of high contention. The main components of comprehensive contention manager are lazy acquisition, extendable timestamp-based conflict detection, and efficient method for capturing conflicts and priorities.

[82] is concerned with implementation issues. [82] considers problems resulting from previous contention management policies like backing off and waiting for time intervals. These strategies make transactions suffer from many aborts that may lead to livelocks, and increased vulnerability to abort because of transactional preemption due to higher priority tasks. Imprecise information and unpredictable benefits resulting from handling long transactions make it difficult to make correct conflict resolution decisions. [82] discriminates between decisions for long and short transactions, as well as, number of threads larger or lower than number of cores. [82] suggests a number of user and kernel level support mechanisms for contention managers, attempting to reduce overhead in current contention managers' implementations. Instead of spin-locks and system calls, the paper uses shared memory segments for communication between kernel and STM library. It also proposes reducing priority of loser threads instead of aborting them. [82] increases time slices for transactions before they are preempted by higher priority threads. This way, long transactions can commit quickly before they are suspended, reducing abort numbers.

For high number of cores, back-off strategies perform poorly. This is due to hot spots created by small set of conflicts. These hotspots repeat in predictable manner. [15] introduces

proactive contention manger that uses history to predict these hotspots and scheduler transactions around them without programmer's input. Proactive contention manager is useful in high contention, but has high cost for low contention. So, [15] uses a hybrid contention managers that begins with back-off strategy for low contention. After a specific threshold for contention level, hybrid contention manager switches to proactive manager.

Contention managers concentrate on preventing starvation through fair policies. They are not suitable for specific systems like real-time systems where stronger behavioural guarantees are required. [51] proposes user-defined priority transactions to make contention management suitable for these specific systems. It investigates the correlation between consistency checking (i.e., finding memory conflicts) and user-defined priority transactions. Transaction priority can be static or dynamic. Dynamic priority increases as abort numbers of transaction increases.

Contention managers are limited in: 1) they are reactive, and suitable only for imminent conflicts. They do not specify when aborted transaction should restart, making them conflict again easily. 2) Contention managers are decentralized because they consume a large part of traffic during high contention. Decentralization prevents global view of the system and limit contention management policy to heuristics. 3) As contention managers are user-level modules, it is difficult to integrate them in HTM. [98] tackles the previous problems by *adaptive transaction scheduling* (ATS). ATS uses contention intensity feedback from the application to adaptively decide number of concurrent transactions running within critical sections. ATS is called only when transaction starts in high contention. Thus, resulting traffic is low and scheduler can be centralized. ATS is integrated into HTM and STM.

[37] presents CAR-STM, a scheduling-based mechanism for STM collision avoidance and resolution. CAR-STM maintains a transaction queue per each core. Each transaction is assigned to a queue by a dispatcher. At the beginning of the transaction, dispatcher uses a conflict probability method to determine the suitable queue for the transaction. The queue with high contention for the current transaction is the most suitable one. All transactions in the same queue are executed by the same thread, thus they are serialized and cannot collide together. CAR-STM uses a serializing contention manager. If one transaction conflicts with another transaction, the former transaction is moved to the queue of the latter. This prevents further collision between them unless the second transaction is moved to a third queue. Thus, CAR-STM uses another serialization strategy in which the two transactions are moved to the third queue. This guarantees conflict between transactions for at most once.

[86] uses HTM to build single and double linked queue, and limited capacity queue. HTM is used as an alternative synchronization operation to CAS and locks. [86] provides worst case time analysis for the implemented data structures. It experimentally compares the implemented data structures with CAS and lock. [86] reverses the role of TM. Transactions are used to build the data structure, instead of accessing data structures inside transactions. [101] presents an implementation for HTM in a Java chip multiprocessor system (CMP).

The used processor is JOP, where worst case execution time analysis is supported.

[10] presents two steps to minimize and limit number of transactional aborts in real-time multiprocessor embedded systems. [10] assumes tasks are scheduled under partitioned EDF. Each task contains at most one transaction. [10] uses multi-versioned STM. In this method, read-only transactions use recent and consistent snapshot of their read sets. Thus, they do not conflict with other transactions and commit on first try. This reduction in abort number comes at the cost of increased memory storage for different versions. [10] uses real-time characteristics to bound maximum number of required versions for each object. Thus, required space is bounded. [10] serializes conflicting transaction in a chronological order. Ties are broken using least laxity and processor identification. [10] does not provide experimental evaluation of its work.

[13] studies the effect of eager versus lazy conflict detection on real-time schedulability. In eager validation, conflicts are detected as soon as they occur. One of the conflicting transactions should be aborted immediately. In lazy validation, conflict detection is delayed to commit time. [13] assumes each task is a complete transaction. [13] proves that synchronous release of tasks does not necessarily lead to worst case response time of tasks. [13] also proves that lazy validation will always result in a longer or equal response time than eager validation. Experiments show that this gap is quite high if higher priority tasks interfere with lower priority ones.

[81]proposes an adaptive scheme to meet deadlines of transactions. This adaptive scheme collects statistical information about execution length of transactions. A transaction can execute in any of three modes depending on its closeness to deadline. These modes are optimistic, visible read and irrevocable. The optimistic mode defers conflict detection to commit time. In visible read, other transactions are informed that a particular location has been read and subject to conflict. Irrevocable mode prevents transaction from aborting. As a transaction gets closer to its deadline, it moves from optimistic to visible read to irrevocable mode. Deadline transactions are supported by the underlying scheduler by disabling preemption for them. Experimental evaluation shows improvement in number of committed transactions without noticeable degradation in transactional throughput.

Chapter 3

Models and Assumptions

We consider a multicore 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. $s_i^k(\theta_1, \theta_2, .., \theta_n)$ is the k^{th} atomic section of τ_i that accesses objects $\theta_1, \theta_2, .., \theta_n$. $s_i^k(\theta)$ 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 [36]. (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 $RC(L)$.

Chapter 4

The ECM and RCM Contention Managers

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 [41].

The rest of this Chapter is organized as follows, Section 4.1 investigates Earliest Deadline Contention Manager under G-EDF scheduling (ECM) and illustrates its behaviour. We provide computations of workload interference and retry cost analysis under ECM. Section 4.2 presents Rate Monotonic Contention Manager under G-RMA scheduling (RCM). It also includes retry cost and response time analysis under ECM. Schedulability of ECM and RCM is compared against schedulability of lock-free in Section 4.3. We conclude the Chapter in Section 4.4.

4.1 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 multicore system scheduled by G-EDF and resolves STM conflicts using the EDF CM. ECM was originally introduced in [45]. 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. [45] assumes the worst case scenario for transactional retry occurs when conflicting transactions are released simultaneously. [45] also assumes all transactions have

the same length. Here, we extend the analysis in [45] 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.

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**
  $s_j^l(\theta)$ aborts;
else
  $s_i^k(\theta)$ aborts;
end

4.1.1 Illustrative Example

Behaviour of ECM can be illustrated by the following example:

- Transaction $s_i^k(\theta) \in \tau_i^x$ begins execution. Currently, $s_i^k(\theta)$ does not conflict with any other transaction.
- Transaction $s_j^l(\theta) \in \tau_j^y$ is released while $s_i^k(\theta)$ is still running. $d_j^y < d_i^x$. So, $p_j^y > p_i^x$. Hence, ECM will abort and restart $s_i^k(\theta)$ in favour of $s_j^l(\theta)$.
- Transaction $s_h^v(\theta) \in \tau_h^u$ is released while $s_j^l(\theta)$ is still running. $d_h^u < d_j^y < d_i^x$. So, $p_h^u > p_j^y > p_i^x$. $s_j^l(\theta)$ and $s_i^k(\theta)$ will abort and retry until $s_h^v(\theta)$ committs.
- $s_h^v(\theta)$ committs. $s_j^l(\theta)$ executes while $s_i^k(\theta)$ aborts and retries.
- $s_j^l(\theta)$ committs. $s_i^k(\theta)$ executes.

4.1.2 G-EDF Interference and workload

The maximum number of times a task τ_j interferes with τ_i is given in [14] and is illustrated in Figure 4.1. Here, the deadline of an instance of τ_j coincides with that of τ_i, and τ_j^1 is delayed by its maximum jitter J_j, which causes all or part of τ_j^1's execution to overlap within T_i. From Figure 4.1, it is seen that τ_j's maximum workload that interferes with τ_i (when there are no atomic sections) in T_i is:

$$\begin{aligned} W_{ij}(T_i) &\leq \left\lfloor \frac{T_i}{T_j} \right\rfloor c_j + min\left(c_j, T_i - \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) \\ &\leq \left\lceil \frac{T_i}{T_j} \right\rceil c_j \end{aligned} \tag{4.1}$$

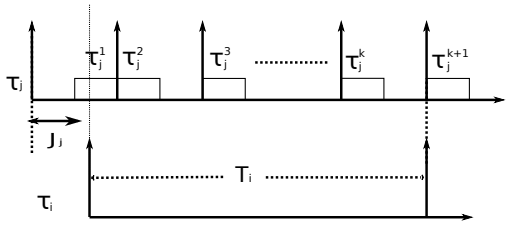


Figure 4.1: Maximum interference between two tasks, running on different processors, under G-EDF

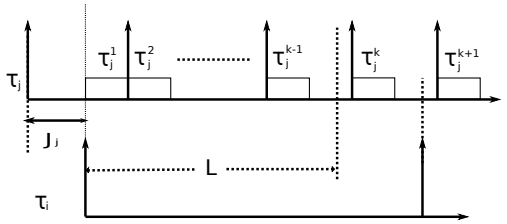


Figure 4.2: Maximum interference during an interval L of T_i

For an interval $L < T_i$, the worst case pattern of interference is shown in Figure 4.2. 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:

$$W_{ij}(L) \leq \left(\left\lceil \frac{L - c_j}{T_j} \right\rceil + 1 \right) c_j \tag{4.2}$$

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

$$W_{ij}(R) = min\left(W_{ij}(R), W_{ij}(T_i)\right) \tag{4.3}$$

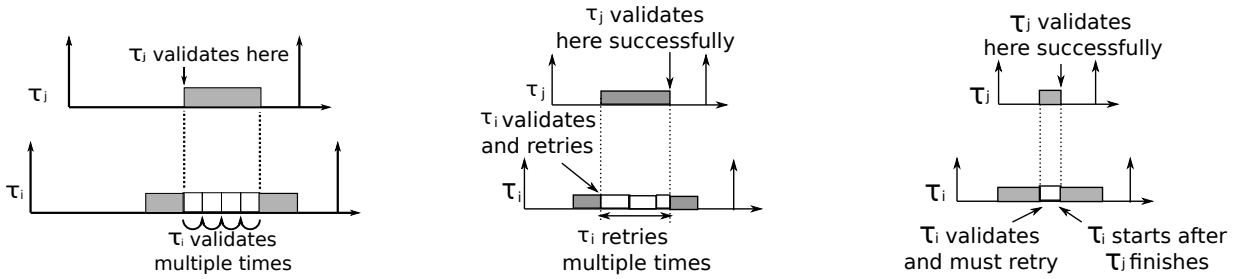
where $W_{ij}(R)$ is calculated by (4.2) if $R < T_i$, otherwise, it is calculated by (4.1).

4.1.3 Retry Cost of Atomic Sections

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

$$RC(T_i) \leq \sum_{\theta \in \theta_i} \left(\left(\sum_{\tau_j \in \gamma_i(\theta)} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta) + s_{max}(\theta)\right) \right) \right) - s_{max}(\theta) + s_{i_{max}}(\theta) \right) \tag{4.4}$$

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



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

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 $s_j^l(\theta)$.

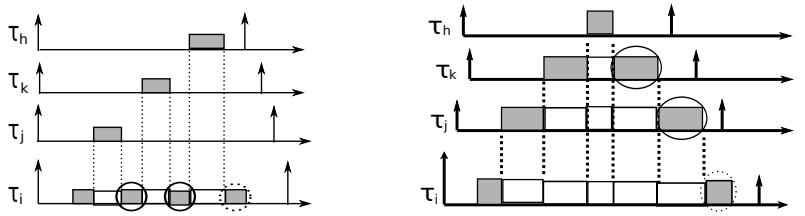
Validation (i.e., conflict detection) in STM is usually done in two ways [85]: a) eager (pessimistic), in which conflicts are detected at access time, and b) lazy (optimistic), in which conflicts are detected at commit time. Despite the validation time incurred (either eager or lazy), $s_i^k(\theta)$ 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 4.3.

In Figure 4.3(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 4.3(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 4.3(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 4.4), 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 4.4(a) and 4.4(b). In Figure 4.4(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



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

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,

$$W_i^p\left(s_j^k\left(\theta\right)\right) \leq len\left(s_j^k\left(\theta\right) + s_{max}\left(\theta\right)\right) \tag{4.5}$$

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:

$$RC\left(T_i\right) \leq \sum_{\theta \in \theta_i} \sum_{\tau_j \in \gamma_i(\theta)} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta) + s_{max}(\theta)\right) \right) \tag{4.6}$$

Here, $\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\left(s_j^l\left(\theta\right) + s_{max}\left(\theta\right)\right)$ 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:*

$$RC(T_i) \leq \sum_{\theta \in \theta_i} min(\Phi_1, \Phi_2) \tag{4.7}$$

where Φ_1 *is calculated by (4.4) for one object* θ *(not the sum of objects in* θ_i*), and*

$$\Phi_2 = \left(\sum_{\tau_j \in \gamma_i(\theta)} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta) + s_{max}^*(\theta)\right) \right) \right) - \bar{s}_{max}(\theta) + s_{i_{max}}(\theta) \tag{4.8}$$

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 (4.4) 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, (4.5) becomes:

$$W_i^p\left(s_j^k(\theta)\right) \leq len\left(s_j^k(\theta) + s^*_{max}(\theta)\right) \tag{4.9}$$

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 4.5, where the maximum atomic section of each task that accesses θ is associated with its corresponding task. According to (4.9), 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.

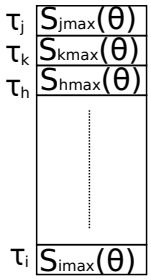


Figure 4.5: Values associated with $s^*_{max}(\theta)$

This means that the subtracted $s_{max}(\theta)$ in (4.4) 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 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 $RC(T_i)$.

4.1.4 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:

$$c_{ji} = c_j - \left(\sum_{\theta \in (\theta_j \wedge \theta_i)} len\left(s_j(\theta)\right) \right) + RC(T_{ji}) \tag{4.10}$$

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 [14], as follows:

$$R_i^{up} = c_i + RC(T_i) + \left\lfloor \frac{1}{m} \sum_{j \neq i} W_{ij}(R_i^{up}) \right\rfloor \tag{4.11}$$

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

$W_{ij}(R_i^{up})$ is calculated by (4.3), and $W_{ij}(T_i)$ is calculated by (4.1), with c_j replaced by c_{ji}, and changing (4.2) as:

$$W_{ij}(L) = max \begin{cases} \left(\left\lceil \frac{L - \left(c_{ji} + \sum_{\theta \in (\theta_j \wedge \theta_i)} len(s_j(\theta)) \right)}{T_j} \right\rceil + 1 \right) c_{ji} \\ \left\lceil \frac{L - c_j}{T_j} \right\rceil . c_{ji} + c_j - \sum_{\theta \in (\theta_j \wedge \theta_i)} len(s_j(\theta)) \end{cases} \tag{4.12}$$

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

Case 1. τ_j^1 (shown in Figure 4.2) 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 (4.12), because they differ in the contribution of their τ_j^1s, and the number of instances after that.

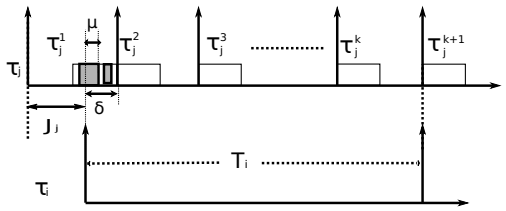


Figure 4.6: Atomic sections of job τ_j^1 contributing to period T_i

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 (4.11). So, (4.7) 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 4.6 can happen.

To counter the situation in Figure 4.6, 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:

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

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, (4.7) can be modified as:

$$RC(T_i) \leq \sum_{\theta \in \theta_i} min \left\{ \begin{array}{l} \left(\left(\sum_{\tau_j \in \gamma_i(\theta)} \lambda_1(j,\theta) + \chi_1(i,j,\theta) \right) \right. \\ \left. - s_{max}(\theta) + s_{i_{max}}(\theta) \right) \\ \left(\left(\sum_{\tau_j \in \gamma_i(\theta)} \lambda_2(j,\theta) + \chi_2(i,j,\theta) \right) \right. \\ \left. - \bar{s}_{max}(\theta) + s_{i_{max}}(\theta) \right) \end{array} \right. \tag{4.13}$$

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, (4.7) becomes:

$$RC(L) \leq \sum_{\theta \in \theta_i} min \begin{cases} \left\{ \begin{array}{l} \left(\sum_{\tau_j \in \gamma_i(\theta)} (\upsilon(L,j)\,\lambda_3(j,\theta))\right) \\ -s_{max}(\theta) + s_{i_{max}}(\theta) \end{array} \right. \\ \left\{ \begin{array}{l} \left(\sum_{\tau_j \in \gamma_i(\theta)} (\upsilon(L,j)\,\lambda_4(j,\theta))\right) \\ -\bar{s}_{max}(\theta) + s_{i_{max}}(\theta) \end{array} \right. \end{cases} \tag{4.14}$$

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

$$RC(R_i^{up}) \leq min \begin{cases} RC(R_i^{up}) \\ RC(T_i) \end{cases} \tag{4.15}$$

where $RC(R_i^{up})$ is calculated by (4.14) if R_i^{up} does not extend to the last interfering instance of τ_j; otherwise, it is calculated by (4.13). The final upper bound on τ_i's response time can be calculated as in (4.11) by replacing $RC(T_i)$ with $RC(R_i^{up})$.

4.2 RCM

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 multicore 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(θ) aborts;
else
    s_i^k(θ) aborts;
end
```

The same illustrative example in Section 4.1.1 is applied for RCM except that tasks' priorities are fixed.

4.2.1 Maximum Task Interference

Figure 4.7 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 4.6, 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.

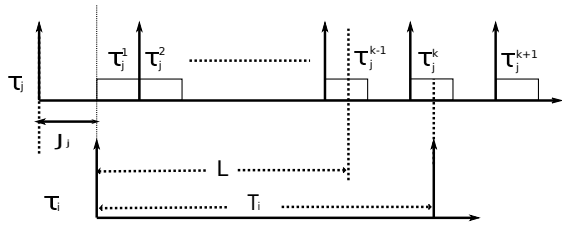


Figure 4.7: Max interference of τ_j to τ_i in G-RMA

Thus, the maximum contribution of τ_j^b to τ_i^a for any duration L can be deduced from Figure 4.7 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.

4.2.2 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:*

$$RC\left(L\right) \leq sum_{\theta \in \theta_i} \left(\left(\sum_{\tau_j^*} \left(\left(\left\lceil \frac{L-c_j}{T_j} \right\rceil + 1 \right) \pi\left(j,\theta\right) \right) \right) - s_{max}^{min}\left(\theta\right) + s_{i_{max}}\left(\theta\right) \right) \tag{4.16}$$

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, (4.14) 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 (4.14), 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 $s^{min}_{max}(\theta)$, 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.

4.2.3 Upper Bound on Response Time

The response time upper bound can be computed using Theorem 7 in [14] with a modification to include the effect of retry cost. The upper bound is given by:

$$R^{up}_i = c_i + RC(R^{up}_i) + \left\lfloor \frac{1}{m} \sum_{j \neq i} W_{ij}(R^{up}_i) \right\rfloor \tag{4.17}$$

where $W_{ij}(R^{up}_i)$ is calculated as in (4.12), c_{ji} is calculated by (4.10), and RC is calculated by (4.16).

4.3 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 [36] is the most relevant to our work.

4.3.1 ECM versus Lock-Free

Claim 4 *For ECM's schedulability to be better or equal to that of [36]'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 (4.15) can be upper bounded as:

$$RC\,(T_i) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s^l_j(\theta)} (2.s_{max}) \right) \right) \tag{4.18}$$

where $s_j^l(\theta)$, $s_{i_{max}}(\theta)$, $s^*_{max}(\theta)$, and $\bar{s}_{max}(\theta)$ 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, (4.18) can be modified as:

$$RC(T_i) = \alpha_{edf}.s_{max} \tag{4.19}$$

The loop retry cost is given by:

$$\begin{aligned} LRC(T_i) &= \sum_{\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right).\beta_{i,j}.r_{max} \\ &= \alpha_{free}.r_{max} \end{aligned} \tag{4.20}$$

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:

$$\begin{aligned} \sum_{\tau_i}\frac{c_i+\alpha_{edf}.s_{max}}{T_i} &\leq \sum_{\tau_i}\frac{c_i+\alpha_{free}.r_{max}}{T_i} \\ \therefore \frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i}\alpha_{free}/T_i}{\sum_{\tau_i}\alpha_{edf}/T_i} \end{aligned} \tag{4.21}$$

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:

$$\begin{aligned} \frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i}(\bar{\alpha}_{free}+\hat{\alpha}_{free})/T_i}{\sum_{\tau_i}\alpha_{edf}/T_i} \\ &= \frac{1}{2} + \frac{\sum_{\tau_i}\hat{\alpha}_{free}/T_i}{\sum_{\tau_i}\alpha_{edf}/T_i} \end{aligned} \tag{4.22}$$

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 (4.22)=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 $T_j \ll T_i$, $\therefore \left\lceil\frac{T_i}{T_j}\right\rceil \to \infty$ and $\zeta_2 \to \infty$. Thus, (4.22)$\to 1/2$.

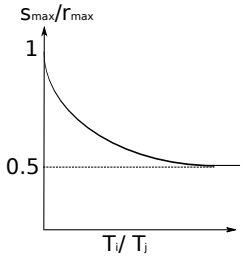


Figure 4.8: Effect of $\left\lceil \frac{T_i}{T_j} \right\rceil$ on $\frac{s_{max}}{r_{max}}$

$\beta_{i,j}$ 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 (4.22)'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 4.8. Claim follows.

4.3.2 RCM versus Lock-Free

Claim 5 *For RCM's schedulability to be better or equal to that of [36]'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 (4.16) is upper bounded by:

$$\sum_{(\tau_j \in \gamma_i) \wedge (p_j > p_i)} \left(\left\lceil \frac{T_i - c_j}{T_j} \right\rceil + 1 \right) .2.\beta_{i,j}.s_{max} \tag{4.23}$$

Consider the same assumptions as in Section 4.3.1. 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:

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{T_i} \alpha_{free}/t\,(T_i)}{\sum_{T_i} \alpha_{rma}/t\,(T_i)} \tag{4.24}$$

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 (4.23) may not have a significant effect on the ratio of (4.24), 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 (4.24) be given indexes k instead of i, and l instead of j. Let tasks in both the numerator and denominator of (4.24) be arranged in the non-increasing priority order, so that $i = k$ and $j = l$. Let α_{free} in (4.24) 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, (4.24) becomes:

$$\begin{aligned} \frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\tau_i} (\bar{\alpha}_{free} + \hat{\alpha}_{free})/T_i}{\sum_{\tau_k} \alpha_{rma}/T_k} \\ &= \frac{1}{2} + \frac{\sum_{\tau_i} \hat{\alpha}_{free}/T_i}{\sum_{\tau_k} \alpha_{rma}/T_k} \end{aligned} \tag{4.25}$$

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 4.9 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.

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 $\beta_{i,j} \rightarrow \infty$, $\therefore$ (4.25)$\rightarrow \infty$. This is

i \ j	1	2	$\cdots$	n
1	0	1	$\cdots$	1
2	0	0	$\ddots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	1
n	0	0	$\cdots$	0

k \ l	1	2	$\cdots$	n
1	0	0	$\cdots$	0
2	1	0		$\vdots$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	0
n	1	$\cdots$	1	0

Figure 4.9: Task association for lower priority tasks than T_i and higher priority tasks than T_k

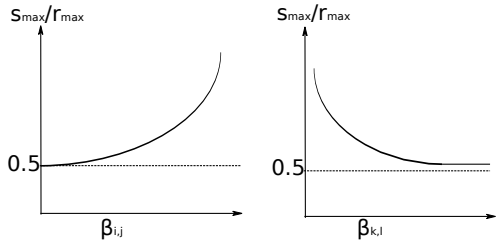
because, $\beta_{i,j}$ represents the number of times a lower priority task τ_j accesses shared objects with a higher priority task τ_i. While this number has a greater effect in lock-free, it does not have any effect under RCM, because lower priority tasks do not affect higher priority ones. Hence, 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 (4.25) $\to 1/2$. 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 4.10(a). If $\beta_{k,l} \to \infty$, then (4.25) $\to 1/2$. 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 (4.25)$\to \infty$. 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 4.10(b).

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 (4.25) $\to 1/2$. 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 4.10(a) and 4.10(b). Claim follows.



(a) (b)

Figure 4.10: Change of s_{max}/r_{max}: a) $\frac{s_{max}}{r_{max}}$ versus $\beta_{i,j}$ and b) $\frac{s_{max}}{r_{max}}$ versus $\beta_{k,l}$

4.4 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 5

The LCM Contention Manager

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) [42]. 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.

The rest of this Chapter is organized as follows: Section 5.1 presents Length-based Contention Manager (LCM) and illustrates its behaviour. Section 5.2 derives LCM properties. Response time analysis of tasks under G-EDF/LCM is given in Section 5.3. Schedulability of G-EDF/LCM is compared to schedulability of ECM and lock-free in Section 5.4. Section 5.5 gives response time analysis for G-RMA/LCM. Schedulability of G-RMA/LCM is compared against RCM and lock-free in Section 5.6. We conclude Chapter in Section 5.7.

5.1 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 (Chapter 4), 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.

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

5.1.1 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. 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:

$$c_{ij}^{kl} = len(s_j^l(\theta))/len(s_i^k(\theta)) \tag{5.1}$$

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).

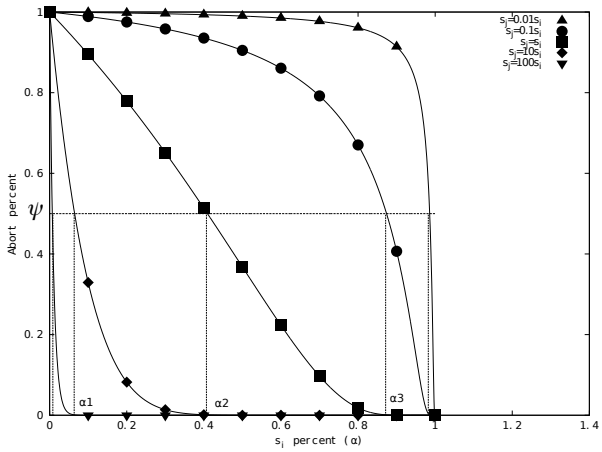


Figure 5.1: Interference of $s_i^k(\theta)$ by various lengths of $s_j^l(\theta)$

The behavior of LCM is illustrated in Figure 5.1. 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 (5.2):

$$f(c_{ij}^{kl}, \alpha) = e^{\frac{-c_{ij}^{kl}\alpha}{1-\alpha}} \tag{5.2}$$

where c_{ij}^{kl} is calculated by (5.1).

Figure 5.1 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 5.1) of its execution length. As $len(s_j^l(\theta))$ decreases, the corresponding α_{ij}^{kl} increases (as shown in Figure 5.1, $\alpha 3 > \alpha 2 > \alpha 1$).

Equation (5.2) 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.

5.1.2 LCM Illustrative Example

Behaviour of LCM can be illustrated by the following example:

- Transaction $s_i^k(\theta) \in \tau_i^x$ begins execution. Currently, $s_i^k(\theta)$ does not conflict with any other transaction.
- Transaction $s_j^l(\theta) \in \tau_j^y$ is released while $s_i^k(\theta)$ is still running. $p_j^y > p_i^x$ (where priority is dynamic in G-EDF, and fixed in G-RMA). $c_{i,j}^{k,l}$, $\alpha_{i,j}^{k,l}$ and α are calculated by steps 4 to 6 in Algorithm 3. $s_i^k(\theta)$ has not reached α percentage of its execution length yet.
- $\alpha < \alpha_{i,j}^{k,l}$. Then, $s_j^l(\theta)$ is allowed to abort and restart $s_i^k(\theta)$.
- $s_j^l(\theta)$ commits. $s_i^k(\theta)$ executes again.
- Transaction $s_h^v(\theta) \in \tau_h^u$ is released while $s_i^k(\theta)$ is running. $p_h^u > p_i^x$. $c_{i,h}^{k,v}$, $\alpha_{i,h}^{k,v}$ and α are calculated by steps 4 to 6 in Algorithm 3. $s_i^k(\theta)$ has already passed α percentage of its execution length. So, $s_h^v(\theta)$ aborts and restarts in favour of $s_i^k(\theta)$.
- Transaction $s_a^b(\theta) \in \tau_a^f$ is released. $p_a^f > p_i^x$ but $p_a^f < p_h^u$. $c_{i,a}^{k,b}$, $\alpha_{i,a}^{k,b}$ and α are calculated by steps 4 to 6 in Algorithm 3. $s_i^k(\theta)$ has not reached α percentage of its execution length yet. So, $s_a^b(\theta)$ is allowed to abort $s_i^k(\theta)$. Because $s_a^b(\theta)$ is just starting, LCM allows $s_h^v(\theta)$ to abort $s_a^b(\theta)$. So, the highest priority transaction is not blocked by an intermediate priority transaction $s_a^b(\theta)$.
- When $s_h^v(\theta)$ commits. $s_a^b(\theta)$ is allowed to execute while $s_i^k(\theta)$ is retrying.
- When $s_a^b(\theta)$ commits, $s_i^k(\theta)$ executes.
- Transaction $s_c^n(\theta) \in \tau_o^z$ is released while $s_i^k(\theta)$ is running. $p_o^z < p_i^x$. So, $s_i^k(\theta)$ commits first, then $s_c^n(\theta)$ is allowed to proceed.

5.2 Properties

LCM properties are given by the following Lemmas. These properties are used to derive retry cost and response time of transactions and tasks under LCM.

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:*

$$W_i^k(s_j^l(\theta)) \leq \alpha_{ij}^{kl} len\Big(s_i^k(\theta)\Big) + len\Big(s_j^l(\theta)\Big) \tag{5.3}$$

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 (5.3). 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:*

$$W_j^l(s_i^k(\theta)) \leq \Big(1 - \alpha_{ij}^{kl}\Big) len\Big(s_i^k(\theta)\Big) \tag{5.4}$$

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 (5.4) 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 (5.2):

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

$\because ln\psi \leq 0$ and $c_{ij}^{kl}, c_{hj}^{kl} > 0, \therefore$ by substitution from (5.1)

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

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

5.3 Response Time of G-EDF/LCM

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

$$RC(T_i) = \left(\sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in \theta_i \wedge \theta_h} \left(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^l(\theta)} len\left(s_h^l(\theta)\right) + \alpha_{max}^{hl} len\left(s_{max}^h(\theta)\right) \right) \right) + \sum_{\forall s_i^y(\theta)} \left(1-\alpha_{max}^{iy}\right) len\left(s_{max}^i(\theta)\right) \quad (5.5)$$

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 $s_i^y(\theta)$.

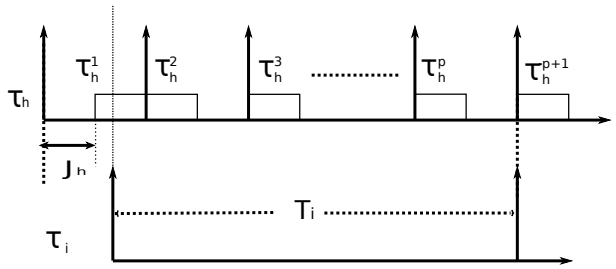


Figure 5.2: τ_h^p has a higher priority than τ_i^x

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 5.2, where one instance of τ_h and τ_h^p coincides with the absolute deadline of τ_i^x.

By using Claims 1, 6, 7, 8 and 9 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 (4.11).

5.4 Schedulability of G-EDF/LCM

We now compare the schedulability of G-EDF/LCM with ECM (Chapter 4) 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:

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

Thus, schedulability is compared by substituting the retry cost added by the synchronization methods in (5.6).

5.4.1 Schedulability of G-EDF/LCM and ECM

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*:*

$$\frac{1-\alpha_{min}}{1-\alpha_{max}} \leq \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil \tag{5.7}$$

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

$$RC(T_i) \leq \sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in (\theta_i \wedge \theta_h)} \left(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^z(\theta)} 2len(s_{max}) \right) \tag{5.8}$$

with the assumption that all lengths of atomic sections of (4.4), (4.8) and (5.5) are replaced by s_{max}. Let α_{max}^{hl} in (5.5) be replaced with α_{max}, and α_{max}^{iy} in (5.5) be replaced with α_{min}. As α_{max}, α_{min}, and $len(s_{max})$ are all constants, (5.5) is upper bounded by:

$$\begin{aligned} RC(T_i) \quad \leq \quad & \left(\sum_{\forall \tau_h \in \gamma_i} \sum_{\forall \theta \in \theta_i \wedge \theta_h} \left(\left\lceil \frac{T_i}{T_h} \right\rceil \sum_{\forall s_h^l(\theta)} (1+\alpha_{max}) \right. \right. \\ & \left. \left. len\Big(s_{max}\Big) \right) \right) + \sum_{\forall s_i^y(\theta)} \Big(1-\alpha_{min}\Big) len\Big(s_{max}\Big) \end{aligned} \tag{5.9}$$

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, (5.8) becomes:

$$RC(T_i) \leq \sum_{\tau_h \in \gamma_i} 2 \left\lceil \frac{T_i}{T_h} \right\rceil \beta_i len(s_{max}) \tag{5.10}$$

and (5.9) becomes:

$$RC(T_i) \leq \beta_i len(s_{max}) \left((1-\alpha_{min}) + \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil (1+\alpha_{max}) \right) \tag{5.11}$$

We can now compare the total utilization of G-EDF/LCM with that of ECM by compar-

ing (5.9) and (5.11) for all τ_i:

$$\sum_{\forall \tau_i} \frac{(1-\alpha_{min}) + \sum_{\forall \tau_h \in \gamma_i} \left(\left\lceil \frac{T_i}{T_h} \right\rceil (1+\alpha_{max}) \right)}{T_i} \leq \sum_{\forall \tau_i} \frac{\sum_{\forall \tau_h \in \gamma_i} 2 \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i} \quad (5.12)$$

(5.12) 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.

5.4.2 G-EDF/LCM versus Lock-free

We consider the retry-loop lock-free synchronization for G-EDF given in [36]. 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 [36]. 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 [36], the retry-loop lock-free algorithm is upper bounded by:

$$RL(T_i) = \sum_{\tau_h \in \gamma_i} \left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta_i r_{max} \quad (5.13)$$

where β_i is as defined in Claim 11. The retry cost of τ_i in G-EDF/LCM is upper bounded by (5.11). By comparing G-EDF/LCM's total utilization with that of the retry-loop lock-free algorithm, we get:

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

$$\therefore \frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_i} \frac{\sum_{\forall \tau_h \in \gamma_i} \left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta_i}{T_i}}{\sum_{\forall \tau_i} \frac{\left((1-\alpha_{min}) + \sum_{\forall \tau_h \in \gamma_i} \left(\left\lceil \frac{T_i}{T_h} \right\rceil (1+\alpha_{max}) \right) \right) \beta_i}{T_i}} \tag{5.14}$$

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:

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

$\therefore$ (5.14) will be:

$$\frac{s_{max}}{r_{max}} \leq 1 + \frac{\sum_{\forall \tau_i} \frac{\omega - 1}{T_i}}{\sum_{\forall \tau_i} \frac{1 + \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{5.15}$$

By substituting the edge values for ω and $\left\lceil \frac{T_i}{T_h} \right\rceil$ in (5.15), we derive that the upper bound on s_{max}/r_{max} lies between 1 and 2.

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

(5.14) becomes

$$\frac{s_{max}}{r_{max}} \leq 0.5 + \frac{\sum_{\forall \tau_i} \frac{\omega - 0.5}{T_i}}{\sum_{\forall \tau_i} \frac{1 + 2 \sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{5.16}$$

By applying the edge values for ω and $\left\lceil \frac{T_i}{T_h} \right\rceil$ in (5.16), we derive that the upper bound on s_{max}/r_{max} lies between 0.5 and 1.

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

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

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

∴ (5.14) becomes:

$$\frac{s_{max}}{r_{max}} \leq 0.5 + \frac{\sum_{\tau_i} \frac{\omega}{T_i}}{2 \sum_{\tau_i} \frac{\sum_{\forall \tau_h \in \gamma_i} \left\lceil \frac{T_i}{T_h} \right\rceil}{T_i}} \tag{5.17}$$

By applying the edge values for ω and $\left\lceil \frac{T_i}{T_h} \right\rceil$ in (5.17), 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, it lies between 0.5 and 1. Claim follows.

5.5 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:*

$$RC\left(T_i\right) = \sum_{\forall \tau_j^*} \left(\sum_{\theta \in (\theta_i \wedge \theta_j)} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \lambda_2(j,\theta) \right) \right) + \sum_{\forall s_i^y(\theta)} \left(1 - \alpha_{max}^{iy}\right) len\left(s_{max}^i(\theta)\right) \tag{5.18}$$

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. (5.3) 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, the Claim follows.

The response time is calculated by (4.17) with replacing $RC(R_i^{up})$ with $RC(T_i)$.

5.6 Schedulability of G-RMA/LCM

5.6.1 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:*

$$\frac{1-\alpha_{min}}{1-\alpha_{max}} \leq \sum_{\forall \tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \tag{5.19}$$

Proof 14 Under the same assumptions as that of Claims 11 and 13, (5.18) can be upper bounded as:

$$RC(T_i) \leq \sum_{\forall \tau_j^*} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) (1+\alpha_{max}) len(s_{max}) \beta_i \right) + (1-\alpha_{min}) len(s_{max}) \beta_i \tag{5.20}$$

For RCM, (4.16) 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:

$$\begin{aligned} & \sum_{\forall \tau_i} \frac{len(s_{max})\beta_i \left((1-\alpha_{min}) + \sum_{\forall \tau_j^*} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) (1+\alpha_{max}) \right) \right)}{T_i} \\ \leq & \sum_{\forall \tau_i} \frac{2len(s_{max})\beta_i \sum_{\forall \tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)}{T_i} \end{aligned} \tag{5.21}$$

(5.21) is satisfied if $\forall \tau_i$ (5.19) is satisfied. Claim follows.

5.6.2 G-RMA/LCM versus Lock-free

Although [36] considers retry-loop lock-free synchronization for G-EDF systems, [36] 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 [36]. 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 (5.18). 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:

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall\tau_i} \frac{\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)}{T_i}}{\sum_{\forall\tau_i} \frac{\left(1-\alpha_{min}\right)+\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)(1+\alpha_{max})}{T_i}} + \frac{2\sum_{\forall\tau_i} \frac{\sum_{\forall\bar{\tau}_j}}{T_i}}{\sum_{\forall\tau_i} \frac{\left(1-\alpha_{min}\right)+\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)(1+\alpha_{max})}{T_i}} \tag{5.22}$$

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:

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

$$\therefore \frac{s_{max}}{r_{max}} \leq 1 + \frac{\sum_{\forall\tau_i} \frac{2\omega_2^i - 1}{T_i}}{\sum_{\forall\tau_i} \frac{1+\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)}{T_i}} \tag{5.23}$$

  As the second term in (5.23) 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.

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

$$\frac{s_{max}}{r_{max}} \leq \frac{1}{2} + \frac{\sum_{\forall\tau_i} \frac{4\omega_2^i - 1}{T_i}}{2\sum_{\forall\tau_i} \frac{1+2\sum_{\tau_j^*}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)}{T_i}} \tag{5.24}$$

  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,

$\omega_2^i \rightarrow n-1$, and $\omega_1^i \rightarrow 0$. Now:

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

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

$$\frac{s_{max}}{r_{max}} \leq \frac{1}{2} + \frac{\sum_{\forall \tau_i} \frac{\omega_2^i}{T_i}}{\sum_{\forall \tau_i} \frac{\sum_{\tau_j^*} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)}{T_i}} \tag{5.25}$$

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, $\omega_2^i \rightarrow n-1$, $\omega_1^i \rightarrow 0$. Now:

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

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.

5.7 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 greater than 0.5. For high values of α in G-RMA/LCM, s_{max}/r_{max} can extend to large values.

Chapter 6

The PNF Contention Manager

In this chapter, we present a novel contention manager for resolving transactional conflicts, called PNF [40]. 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.

The rest of this Chapter is organized as follows: Section 6.1 discusses limitations of previous contention managers and the motivation to PNF. Section 6.2 give a formal description of PNF. Section 6.3 derives PNF's properties. We upper bound retry cost and response time under PNF in Section 6.4. Schedulability comparison between PNF and previous synchronization techniques is given in Section 6.5. We conclude Chapter in Section 6.6.

6.1 Limitations of ECM, RCM, and LCM

ECM, RCM and LCM [41, 42] assumes that each transaction accesses only one object. This assumption simplifies the retry cost (Claims 2, 3, 10 and 13) and response time analysis (Sections 4.1, 4.2, 5.3 and 5.5). Besides, it enables a one-to-one comparison with lock-free synchronization in [36]. With multiple objects per transaction, ECM, RCM and LCM will face 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 Chapters 4 and 5 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 Section 6.2.

6.2 The PNF Contention Manager

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

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

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).

Algorithm 4: PNF

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[2] then
14                 Restore priority of task owning n(z);
15                 Assign n(z) as executing transaction;
16                 Add n(z) to m-set and remove it from n-set;
17                 Select n(z) for commit;
18             else
19                 Wait until processor available
20             end
21         end
22         move to the next n(z);
23     end
24 end
```

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. Transactions in the n_set are known as *retrying transaction*.It also holds transactions that cannot currently execute, because processors are busy, either

[2]An idle processor or at least one that runs a non-atomic section task with priority lower than the task holding $n(z)$.

due to processing executing transactions 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 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. 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, all preempted and non-preempted transactions in the n-set are examined when any of the executing transaction commits. Then, one or more transactions are selected from the n-set to be executing transactions. If a retrying transaction is selected as an executing transaction, the task that owns the retrying transaction regains its priority.

When a new transaction is released, and if it does not conflict with any of the executing transactions (Step 1), then it will allocate a slot in the m-set and becomes an executing transaction. When this transaction is released (i.e., its containing task is already allocated to a processor), it will be able to access a processor immediately. This 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 executing transactions.

When one of the executing transactions 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 to the lowest priority (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) ("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 its original priority is restored. If s_h^b is added to the m-set, the new m-set is compared 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 term "P" in the algorithm name). 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.

6.2.1 Illustrative Example

We illustrate PNF with an example. We use the following notions: $s_a^b \in \tau_a^k$ is transaction s_a^b in job τ_a^k. $s_a^b(\theta_1, \theta_2, \theta_3)$ means that s_a^b accesses objects $\theta_1, \theta_2, \theta_3$. $p(s_a^b)$ is the priority of transaction s_a^b. p_i^j is the priority of job τ_i^j. If $s_a^b \in \tau_a^j$, $\therefore p_o(s_a^b) = p_a^j$, where $p_o(s_a^b)$ is the original priority of s_a^b. $p(s_a^b) = -1$, if s_a^b is a retrying transaction; $p(s_a^b) = p_o(s_a^b)$ otherwise. m-set= $\{s_a^b, s_i^k\}$ means that the m-set contains transactions s_a^b and s_i^k regardless of their order. n-set= $\{s_a^b, s_i^k\}$ means that the n-set contains transactions s_a^b and s_i^k in that order, where $p_o(s_a^b) > p_o(s_i^k)$. m-set (n-set) = $\{\phi\}$ means that m-set (n-set) is empty. Assume there are five processors.

1. Initially, m-set= n-set= $\{\phi\}$. $s_a^b(\theta_1, \theta_2) \in \tau_a^b$ is released and checks m-set for conflicting transactions. As m-set is empty, s_a^b finds no conflict and becomes an executing transaction. s_a^b is added to m-set. m-set= $\{s_a^b\}$ and n-set= $\{\phi\}$. s_a^b is executing on processor 1.
2. $s_c^d(\theta_3, \theta_4) \in \tau_c^d$ is released and checks m-set for conflicting transactions. s_c^d does not conflict with s_a^b as they access different objects. s_c^d becomes an executing transaction and is added to m-set. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{\phi\}$. s_c^d is executing on processor 2.
3. $s_e^f(\theta_1, \theta_5) \in \tau_e^f$ is released and $p_o(s_e^f) < p_o(s_a^b)$. s_e^f conflicts with s_a^b when it checks m-set. s_e^f is added to n-set and becomes a retrying transaction. $p(s_e^f)$ becomes -1. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_e^f\}$. s_e^f is retrying on processor 3.
4. $s_g^h(\theta_1, \theta_6) \in \tau_g^h$ is released and $p_o(s_g^h) > p_o(s_a^b)$. s_g^h conflicts with s_a^b. Though s_g^h is of higher priority than s_a^b, s_a^b is an executing transaction. So s_a^b runs non-preemptively. s_g^h is added to n-set before s_e^f, because $p_o(s_g^h) > p_o(s_e^f)$. $p(s_g^h)$ becomes -1. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_g^h, s_e^f\}$. s_g^h is retrying on processor 4.
5. $s_i^j(\theta_5, \theta_7) \in \tau_i^j$ is released. $p_o(s_i^j) < p_o(s_e^f)$. s_i^j does not conflict with any transaction in m-set. Though s_i^j conflicts with s_e^f and $p_o(s_i^j) < p_o(s_e^f) < p_o(s_g^h)$, s_e^f and s_g^h are retrying transactions. s_i^j becomes an executing transaction and is added to m-set. m-set= $\{s_a^b, s_c^d, s_i^j\}$ and n-set= $\{s_g^h, s_e^f\}$. s_i^j is executing on processor 5.
6. τ_k^l is released. τ_k^l does not access any object. $p_k^l < p_o(s_e^f) < p_o(s_g^h)$, but $p(s_e^f) = p(s_g^h) = -1$. Since there are no more processors, τ_k^l preempts τ_e^f, because the currently assigned priority to $\tau_e^f = p(s_e^f) = -1$ and $p_o(s_g^h) > p_o(s_e^f)$. τ_k^l is running on processor 3. This way, PNF optimizes processor usage. The m-set and n-set are not changed. Although s_e^f is preempted, n-set still records it, as s_e^f might be needed (as will be shown in the following steps).
7. s_i^j commits. s_i^j is removed from m-set. Transactions in n-set are checked from the first (highest p_o) to the last (lowest p_o) for conflicts against any executing transaction. s_g^h is checked first because $p_o(s_g^h) > p_o(s_e^f)$. s_g^h conflicts with s_a^b, so s_g^h cannot be an executing transaction. Now it is time to check s_e^f, even though s_e^f is preempted in step 6. s_e^f also conflicts with s_a^b, so s_e^f cannot be an executing transaction. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_g^h, s_e^f\}$. Now, s_e^f can be retrying on processor 5 if τ_i^j has finished execution.

Otherwise, τ_i^j continues running on processor 5 and s_e^f is still preempted. This is because, $p(s_e^f) = -1$ and $p_i^j > p(s_e^f)$. Let us assume that τ_i^j is still running on processor 5.

8. s_a^b commits. s_a^b is removed from m-set. Transactions in n-set are checked as done in step 7. s_g^h does not conflict with any executing transaction any more. s_g^h becomes an executing transaction. s_g^h is removed from n-set and added to m-set, so m-set= $\{s_c^d, s_g^h\}$. Now, s_e^f is checked against the new m-set. s_e^f conflicts with s_g^h, so s_e^f cannot be an executing transaction. s_e^f can be retrying on processor 1 if τ_a^b has finished execution. Otherwise, s_e^f remains preempted, because $p(s_e^f) = -1$ and $p_a^b > p(s_e^f)$. n-set= $\{s_e^f\}$. Let us assume that τ_a^b is still running on processor 1.
9. s_g^h commits. s_g^h is removed from m-set. τ_g^h continues execution on processor 4. Transactions in n-set are checked again. s_e^f is the only retrying transaction in the n-set, and it does not conflict with any executing transactions. Now, the system has τ_a^b running on processor 1, s_c^d executing on processor 2, τ_k^l running on processor 3, τ_g^h running on processor 4, and τ_i^j running on processor 5. s_e^f can become an executing transaction if it can find a processor. Since p_i^j, $p_k^l < p_o(s_e^f)$, s_e^f can preempt the lowest in priority between τ_i^j and τ_k^l. s_e^f now becomes an executing transaction. s_e^f is removed from the n-set and added to the m-set. So, m-set= $\{s_c^d, s_e^f\}$ and n-set= $\{\phi\}$. If p_i^j, p_k^l were of higher priority than $p_o(s_e^f)$, then s_e^f would have remained in n-set until a processor becomes available.

The example shows that PNF avoids transitive retry. This is illustrated in step 5, where $s_i^j(\theta_5, \theta_7)$ is not affected by the retry of $s_e^f(\theta_1, \theta_5)$. The example also explains how PNF optimizes processor usage. This is illustrated in step 6, where the retrying transaction s_e^f is preempted in favor of τ_k^l.

6.3 Properties

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

Proof 17 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 due to s_z^h. Note that s_z^h and s_j^l cannot exit together in the m-set as they have shared objects. But they both can be in the n-set, as they 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 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.

6.4 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 *Under PNF, the maximum retry cost suffered by a transaction s_i^k due to a transaction 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:*

$$RC(L) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^l}(\theta)} len\left(\bar{s_j^l}(\theta) \right) \right) \right) \tag{6.1}$$

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 executing 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 executing transactions. 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 (6.1) 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:*

$$D(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h}\right) \right) \right\rfloor \tag{6.2}$$

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, executing transactions are non preemptive. So, a lower priority executing 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 (6.1).

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 (6.1). Besides, (6.1) does not divide the retry cost by m as done in (6.2). Thus, the worst case scenario requires inclusion of s_j^h in (6.1), and not in (6.2).

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.

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 7 in [14].

Claim 23 *The response time of a job τ_i^x, during an interval $L \leq T_i$, under PNF/G-EDF is upper bounded by:*

$$R_i^{up} = c_i + RC(L) + D_{edf}(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} W_{ij}(R_i^{up}) \right\rfloor \tag{6.3}$$

where $RC(L)$ is calculated by (6.1). $D_{edf}(\tau_i^x)$ is the same as $D(\tau_i^x)$ defined in (6.2). However, for G-EDF systems. $D_{edf}(\tau_i^x)$ is calculated as:

$$D_{edf}(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \tau_j^l} \begin{cases} 0 & , L \leq T_i - T_j \\ \sum_{\forall \ddot{s}_j^h} len\left(\ddot{s}_j^h\right) & , L > T_i - T_j \end{cases} \right\rfloor \tag{6.4}$$

and $W_{ij}(R_i^{up})$ is calculated by (4.3).

Proof 23 Response time for τ_i^x is calculated by (4.3) 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 (6.4). 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 (6.2) is replaced with 1 in the second case in (6.4). Claim follows.

Claim 24 *The response time of a job τ_i^x, during an interval $L \leq T_i$, under PNF/G-RMA is upper bounded by:*

$$R_i^{up} = c_i + RC(L) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i, p_j > p_i} W_{ij}(R_i^{up}) \right\rfloor \tag{6.5}$$

where $RC(L)$ is calculated by (6.1), $D(\tau_i^x)$ is calculated by (6.2), and $W_{ij}(R_i^{up})$ is calculated by (4.2).

Proof 24 Proof is same as of Claim 23, except that G-RMA assigns fixed priorities. Hence, (6.2) can be used directly for calculating $D(\tau_i^x)$ without modifications. Claim follows.

6.5 PNF vs. Competitors

We now (formally) compare the schedulability of G-EDF (G-RMA) with PNF against ECM, RCM, LCM and lock-free synchronization [36, 41, 42]. 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:

$$\begin{aligned} \sum_{\forall \tau_i} \frac{c_i + RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{c_i + RC_B(T_i)}{T_i} \\ \therefore \sum_{\forall \tau_i} \frac{RC_A(T_i)}{T_i} &\leq \sum_{\forall \tau_i} \frac{RC_B(T_i)}{T_i} \end{aligned} \tag{6.6}$$

As described in Section 6.1, the set of common objects needs to be extended under PNF's competitors. 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 discussed in Section 6.1. 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}. i.e., 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 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:*

$$RC_{to}(L) = RC(L) + RC_{re}(L) \tag{6.7}$$

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 (4.15) for ECM and (5.5) 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:

$$RC_{re}(L) = \sum_{\forall \tau_j \in \zeta_i} \begin{cases} \left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} & , L \leq T_i - T_j \\ \\ \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{i_{max}} & , L > T_i - T_j \end{cases} \tag{6.8}$$

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 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 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 [14]) 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 [14]) 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 $len(s_{i_{max}})$. The total retry cost suffered by τ_i^x is the combination of RC and RC_{re}.

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:*

$$RC_{to}(L) = RC(L) + RC_{re}(L) \tag{6.9}$$

where $RC(L)$ and $RC_{re}(L)$ are defined in Claim 25. $RC(L)$ is calculated by (4.16) for RCM, and (5.18) for G-RMA/LCM. $RC_{re}(L)$ is calculated by:

$$RC_{re}(L) = \sum_{\forall \tau_j \in \zeta_i^*} \left(\left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} \right) \tag{6.10}$$

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 (6.8), where $s_{i_{max}}$ is replaced by $r_{i_{max}}$. RC_{re} under G-RMA with lock-free synchronization is calculated by (6.10), 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. The pattern is also the same in RCM, G-RMA/LCM, and G-RMA with lock-free.

6.5.1 PNF versus ECM

Claim 28 *In the absence of transitive retry, PNF/G-EDF's schedulability is better or equal to ECM's when conflicting atomic sections have equal lengths.*

Proof 28 Substitue $RC_A(T_i)$ and $RC_B(T_i)$ in (6.6) with (6.1) and (6.7), 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:

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

where RC_{re} is given by (6.8). $g(\tau_i)$ includes effect of transitive retry. Let:

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

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

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

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 (6.6), we get:

$$\sum_{\forall\tau_i} \frac{\eta_1(\tau_i)}{T_i} \leq \sum_{\forall\tau_i} \frac{\eta_2(\tau_i) + g(\tau_i)}{T_i} \tag{6.11}$$

Assume that $g(\tau_i)_{\forall\tau_i} \rightarrow 0$. From (6.11), we note that by keeping every $len(s_j^{k^-}(\theta)) \leq len(s_{max}^j(\theta))$ for each τ_i, $\tau_j \in \gamma_i$, and $\theta \in \theta_i$, (6.11) holds. Due to G-EDF's dynamic priority, $s_{max}^j(\theta)$ can belong to any task other than τ_j. By keeping $len(s_j^{k^-}(\theta)) \leq len(s_{max}^j(\theta))$, then 6.11 holds. By generalizing this condition to any $s_j^k(\theta)$ and $s_{max}^j(\theta)$, then 6.11 holds if all atomic sections in all tasks have equal lengths. Claim follows.

6.5.2 PNF versus RCM

Claim 29 *In the absence of transitive retry, PNF/G-RMA's schedulability is better or equal to RCM's schedulability when a large number of tasks heavily conflict. PNF's schedulability is improved compared with RCM's, when atomic section length increases as priority increases.*

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 (6.6) with (6.1) and (6.9), respectively. Let:

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

where RC_{re} and ζ_i^* are defined by (6.10). $g(\tau_i)$ includes effect of transitive retry. 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:

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

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

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

By substitution of $g(\tau_i)$, $\eta_1(\tau_i)$, $\eta_2(\tau_i)$, and $\eta_3(\tau_i)$ in (6.6):

$$\sum_{\forall\tau_i} \frac{\eta_1(\tau_i) + \eta_2(\tau_i)}{T_i} \leq \sum_{\forall\tau_i} \frac{\eta_3(\tau_i) + g(\tau_i)}{T_i} \tag{6.12}$$

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:

$$\eta_2(\tau_i) = 2 \sum_{\forall \tau_j \in (\gamma_i \cap \bar{\zeta_i})} \sum_{\forall \theta \in \theta_i} \sum_{\forall \bar{s_j^k}(\theta)} len\left(\bar{s_j^k}(\theta)\right) \tag{6.13}$$

Let:

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

By substitution of (6.13) and subtraction of $\sum_{\forall \tau_i} \frac{\eta_1(\tau_i)}{T_i}$ from both sides of (6.12), we get:

$$2 \sum_{\forall \tau_i} \frac{\eta_2(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_4(\tau_i) + g(\tau_i)}{T_i} \tag{6.14}$$

Assume that $g(\tau_i)_{\forall \tau_i} \rightarrow 0$. From (6.14), we note that when higher priority jobs increasingly conflict with lower priority jobs, (6.14) tends to hold. (6.14) also tends to hold if $len(\bar{s_{max}^j}(\theta))$ in the right hand side of (6.14) is larger than $len(\bar{s_j^k}(\theta))$ in the left hand side of (6.14), which means atomic section length increases as priority increases. Claim follows.

6.5.3 PNF versus G-EDF/LCM

Claim 30 *In the absence of transitive retry, PNF/EDF's schedulability is equal or better than G-EDF/LCM's if the conflicting atomic section lengths are approximately equal and all α terms approach 1.*

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

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

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

where α_{max}^{jl} is defined in (5.5). Following the same steps in the proof of Claim 28, we get:

$$\sum_{\forall \tau_i} \frac{\eta_1(\tau_i)}{T_i} \leq \sum_{\forall \tau_i} \frac{\eta_2(\tau_i) + g(\tau_i)}{T_i} \tag{6.15}$$

Assume that $g(\tau_i)_{\forall\tau_i} \rightarrow 0$. Thus, we ignore the effect of transitive retry and retry cost due to the 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 (6.15). Then, PNF/EDF's schedulability equals LCM/EDF's schedulability if $\left\lceil \frac{T_i}{T_j} \right\rceil = 1, \forall\tau_i, \tau_j$ (which means equal periods for all tasks). If $\left\lceil \frac{T_i}{T_j} \right\rceil > 1, \forall\tau_i, \tau_j$, PNF/EDF's schedulability is better than LCM/EDF's. PNF/EDF's schedulability becomes more better than LCM/EDF's schedulability if $g(\tau_i)$ is not zero. Claim follows.

6.5.4 PNF versus G-RMA/LCM

Claim 31 *In the absence of transitive retry, PNF's schedulability is equal or better than G-RMA/LCM's if: 1) lower priority tasks suffer increasing number of conflicts from higher priority tasks, 2) the lengths of the atomic sections increase as task priorities increase, and 3) α terms increase.*

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

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

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

Following the steps of Claim 29's proof, $\therefore$(6.6) becomes:

$$2 \sum_{\forall\tau_i} \frac{\eta_2(\tau_i)}{T_i} \leq \sum_{\forall\tau_i} \frac{\eta_4(\tau_i) + g(\tau_i)}{T_i} \tag{6.16}$$

Assume that the effect of transitive retry and retry cost due to the release of higher priority jobs is negligible ($g(\tau_i) \rightarrow 0$). (6.16) holds if: 1) the 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 (6.16); 2) α terms approach 1; and 3) the lengths of the atomic sections increase as priority increases. This makes $len(s_{max}^j(\theta))$ in (6.16)'s right side to be greater than $len(s_j^{\bar{k}}(\theta))$ in (6.16)'s left side. Claim follows.

6.5.5 PNF versus Lock-free Synchronization

Lock-free synchronization [36, 41] 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 (6.6) is replaced with:

$$\sum_{\forall \tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} r_{max} \right) + RC_{re}(T_i) \tag{6.17}$$

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 [36]. r_{max} is the maximum execution cost of a single iteration of any retry loop of any task [36]. $RC_{re}(T_i)$ is defined in Claim 27. Lock-free synchronization does not depend on priorities of tasks. Thus, (6.17) 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 [36]. 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 32 is made to enable a comparison between PNF and lock-free. Let $RC_A(T_i)$ in (6.6) be replaced with (6.1) and $RC_B(T_i)$ be replaced with (6.17). To simplify comparison, (6.1) 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, $\beta^*_{i,j} = \beta_{i,j}$, and (6.6) will be:

$$\sum_{\forall \tau_i} \frac{\sum_{\tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} s_{max} \right)}{T_i} \leq \sum_{\forall \tau_i} \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{i,j} r_{max} + RC_{re}(\tau_i) \right)}{T_i} \tag{6.18}$$

From (6.18), we note that if $s_{max} \leq r_{max}$, then (6.18) holds.

6.6 Conclusion

Transitive retry increases transactional retry cost under ECM, RCM, and LCM. PNF avoids transitive retry by avoiding transactional preemptions. PNF reduces the priority of aborted transactions to enable other tasks to execute, increasing processor usage. Executing transactions are not preempted due to the release of higher priority jobs. On the negative side of PNF, higher priority jobs can be blocked by executing transactions of lower priority jobs.

EDF/PNF's schedulability is equal or better than ECM's when atomic section lengths are almost equal. RMA/PNF's schedulability is equal or better than RCM's when lower priority jobs suffer greater conflicts from higher priority ones. Similar conditions hold for the schedulability comparison between PNF and LCM, in addition to the increase of α terms to 1. This is logical as LCM with G-EDF (G-RMA) defaults to ECM (RCM) with $\alpha \rightarrow 1$. For PNF's schedulability to be equal or better than lock-free, the upper bound on s_{max}/r_{max} must be 1, instead of 0.5 under ECM and RCM.

Chapter 7

Implementation and Experimental Evaluations

Having established upper bounds for retry cost of different contention managers, and the conditions under which each one is prefered. We now would like to understand how each CM retries in practice (i.e., on average) compared with that of competitor methods. Since this can only be understood experimentally, we implement ECM, RCM, LCM, PNF and lock-free and conduct experimental studies.

The rest of this Chapter is organized as follow: Section 7.1 outlines the experimental settings and used task sets for comparing different contention managers and lock-free. Section 7.2 discusses experimental results.

7.1 Experimental Setup

We used the ChronOS real-time Linux kernel [35] and the RSTM library [84]. We modified RSTM to include implementations of ECM, RCM, LCM, and PNF 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 $0.0292546250\ \mu s$.

We used 3 sets of 4, 8 and 20 tasks. The structure of these tasks are shown in Table 7.1. 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. As lock-free cannot access more than one object in one atomic operation, tasks share one object per transaction when lock-free is included in comparison. Then, CMs are compared against each other discarding lock-free.

Table 7.1: 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

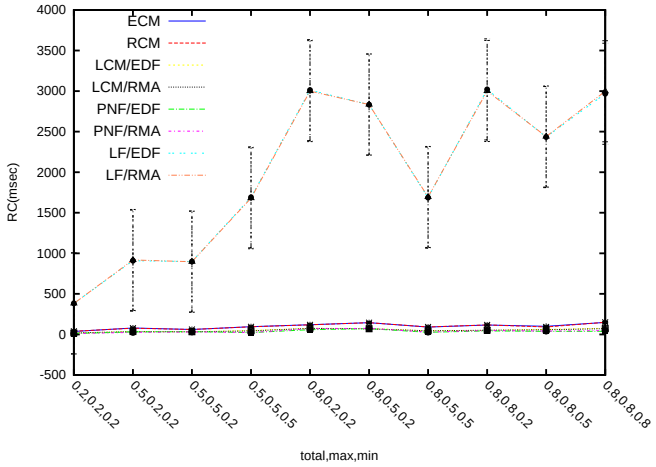
The difficulty in testing with PNF is to incur transitive retry cases. Tasks are arranged in non-decreasing order of periods, and each task shares objects only with the previous and next tasks. Each task begins with an atomic section. Thus, increasing the opportunity of transitive retry.

7.2 Results

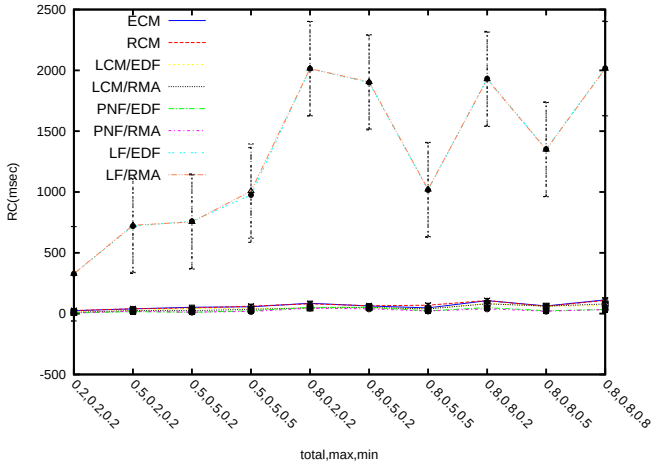
Figure 7.1 shows average retry cost under ECM, RCM, LCM, PNF and lock-free for each task set. Figure 7.2 shows average retry cost for only contention managers for each task set. The x-axis has three parameters a, b, c. a specifies the relative total length of all atomic sections to the length of the task. b specifies the maximum relative length of any atomic section to the length of the task. c specifies the minimum relative length of any atomic section to the length of the task. Each data point in the figure has a confidence level of 0.95. Only one object per transaction is shared in Figures 7.1 and 7.2.

Lock-free is the longest technique as it provides no conflict resolution. PNF better or comparable retry cost than ECM, RCM and LCM. As we move from 4 to 8 to 20 task set, retry costs of different contention managers get closer to each other. This is explained by noting that each task set in Table 7.1 is organized in non-decreasing order of periods, and c_i/T_i for almost each τ_i is low. Besides, each task shares objects only with the previous and next tasks, and tasks are released at the same time to enforce transitive retry. While the first instances of all tasks have a high potential of conflict, the contention level decreases with time for higher number of tasks. Thus, for the 20 task set, contention level is the lowest. Hence, retry costs of all contention managers get closer as number of tasks increases.

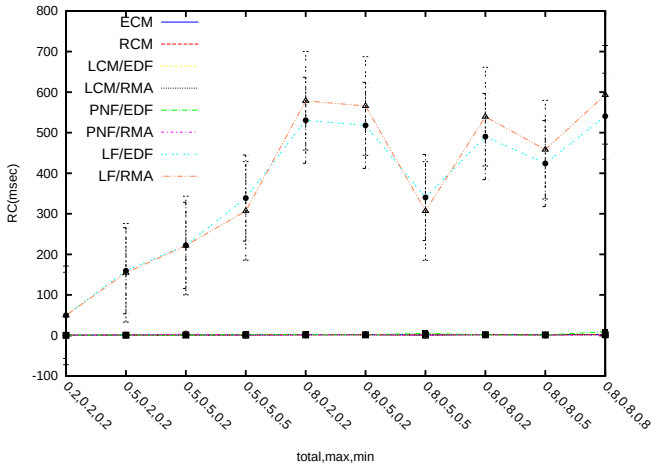
We compared retry cost for different contention managers with multiple objects per transaction and different levels of read/writer operations. Figure 7.3 shows retry cost of the three task sets sharing 20 objects per transaction, with 40% write operations and 60% read operations. The same experiment is repeated in Figure 7.4 with 80% write operations, and 20% read operations. Figure 7.5 repeats the same experiment with 100% write operations. The same previous three experiments were repeated in Figures 7.6, 7.7 and 7.8 with 40 objects per transaction. Figures 7.3 to 7.8 show consistent trends with Figure 7.2 except that retry cost of PNF is shorter than the others even with increasing number of tasks. For the 20 task set, PNF retry cost is a little shorter than LCM, but much better than ECM and RCM. This happens because of sharing multiple objects per transaction. Thus, contention level is increased than in sharing 1 object per transaction. Besides, transitive retry exists which makes PNF better than the others.



(a) 4 tasks

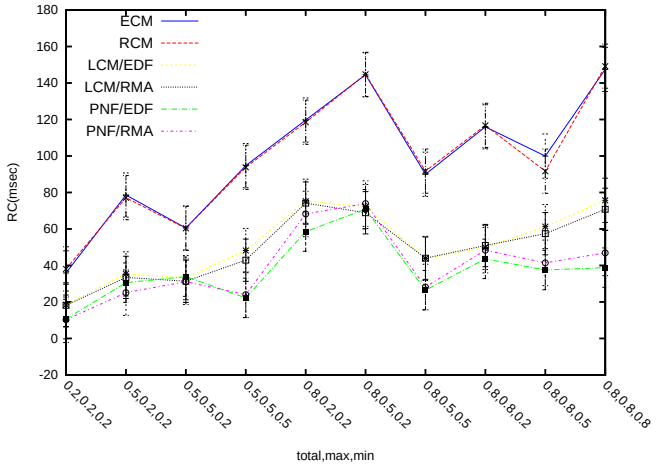


(b) 8 tasks

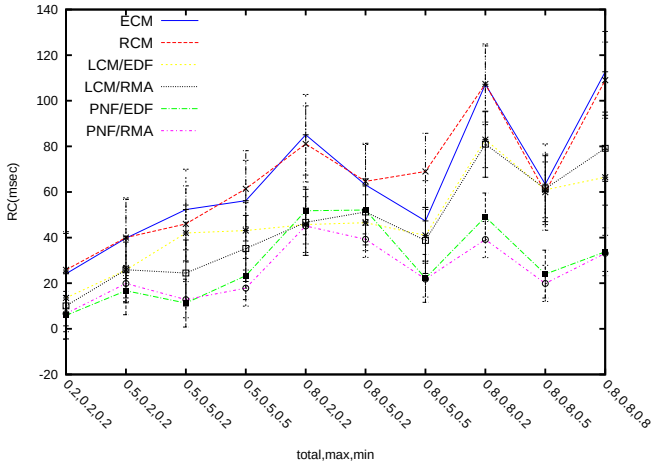


(c) 20 tasks

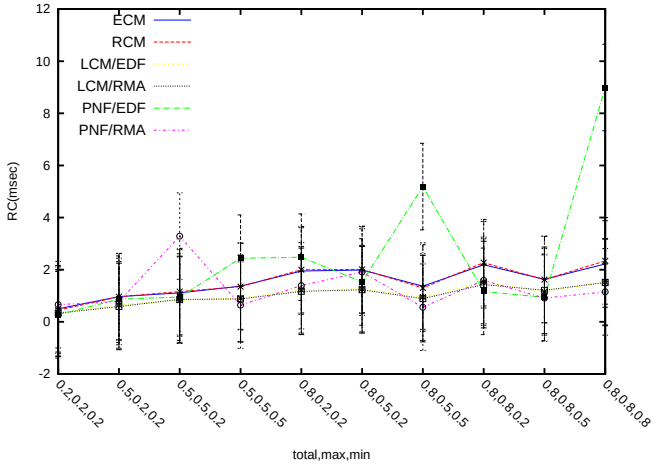
Figure 7.1: Average retry cost for 1 object per transaction for different values of total, maximum and minimum atomic section length under all synchronization techniques



(a) 4 tasks

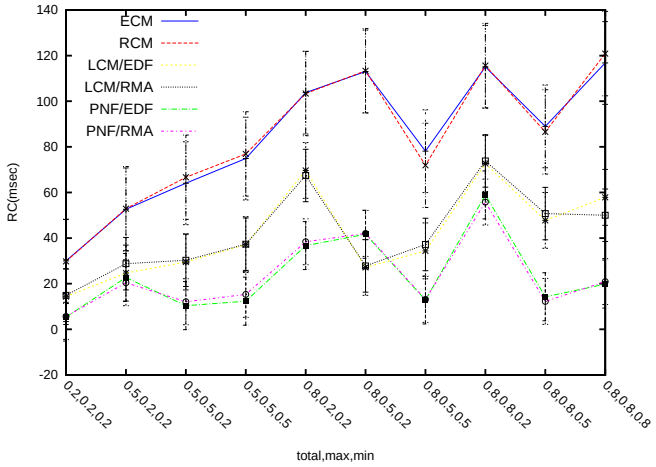


(b) 8 tasks

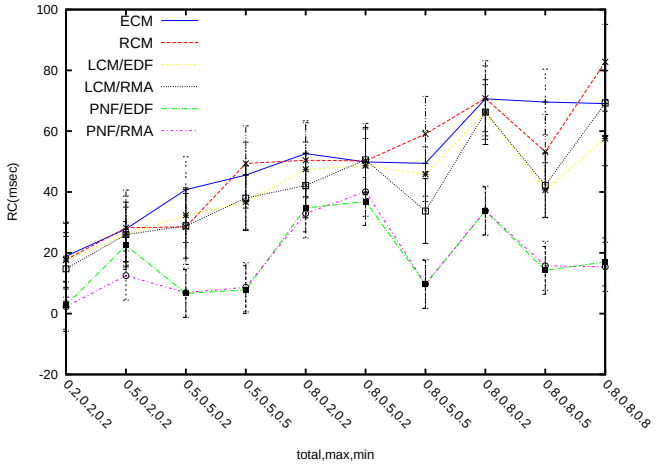


(c) 20 tasks

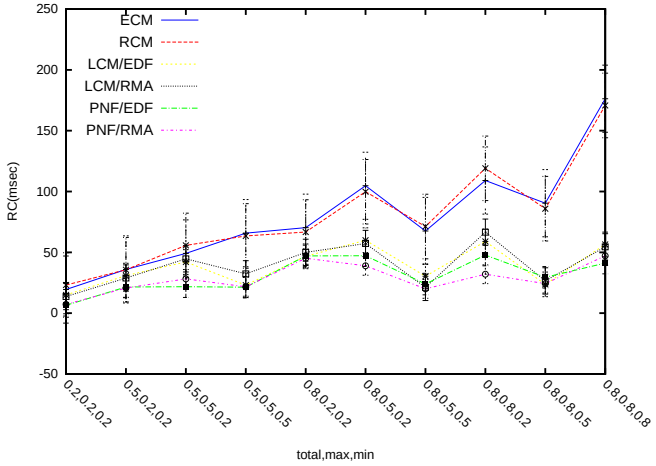
Figure 7.2: Average retry cost for 1 object per transaction for different values of total, maximum and minimum atomic section length under contention managers only



(a) 4 tasks

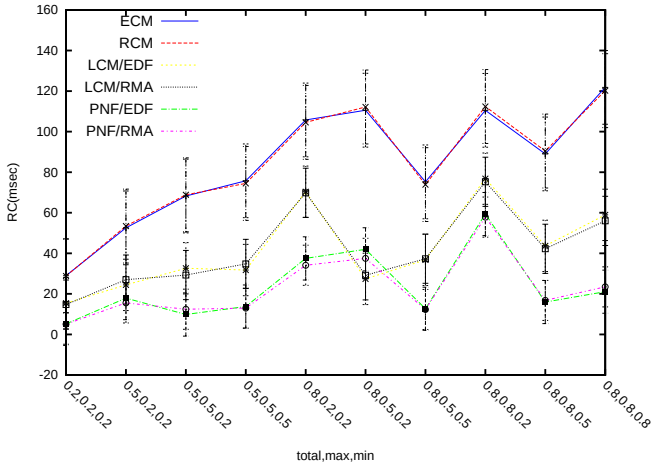


(b) 8 tasks

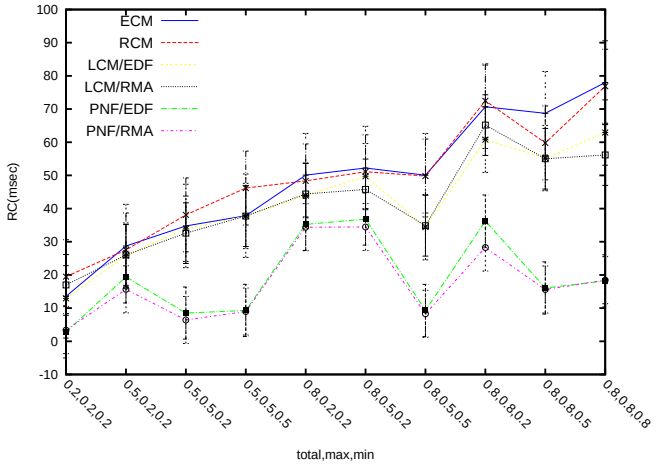


(c) 20 tasks

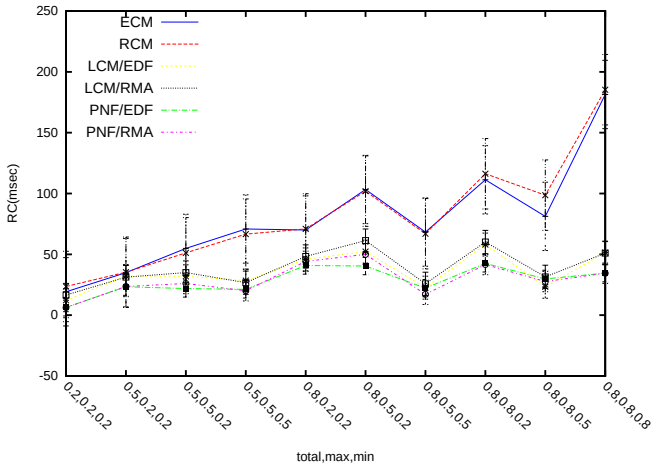
Figure 7.3: Average retry cost for 20 objects per transaction, 40% write operations for different values of total, maximum and minimum atomic section length under different CMs



(a) 4 tasks

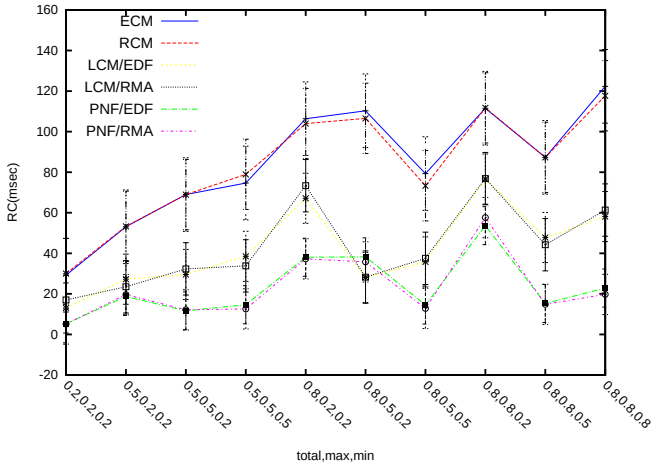


(b) 8 tasks

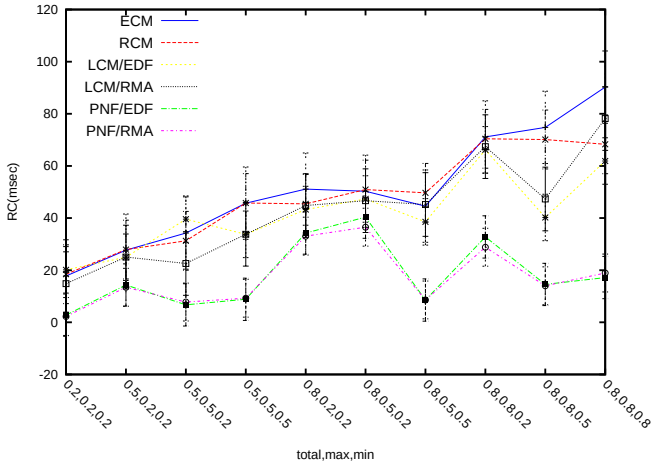


(c) 20 tasks

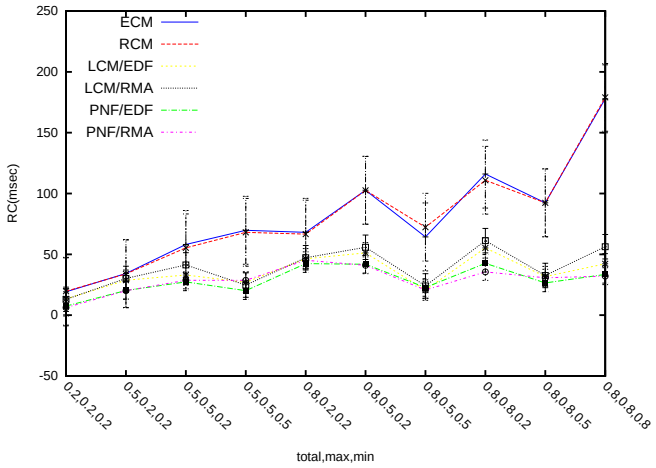
Figure 7.4: Average retry cost for 20 objects per transaction, 80% write operations for different values of total, maximum and minimum atomic section length under different CMs



(a) 4 tasks

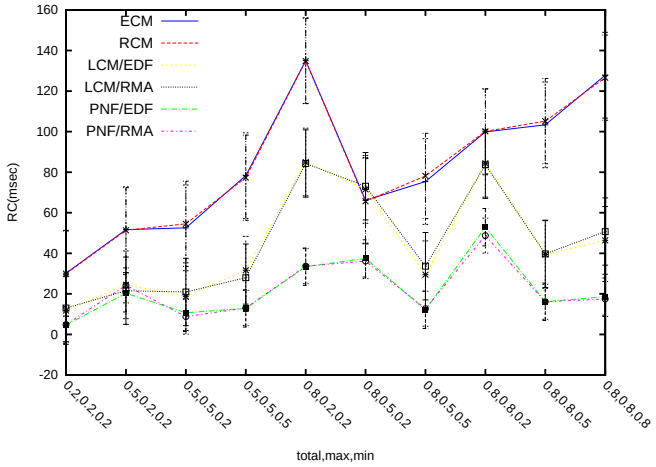


(b) 8 tasks

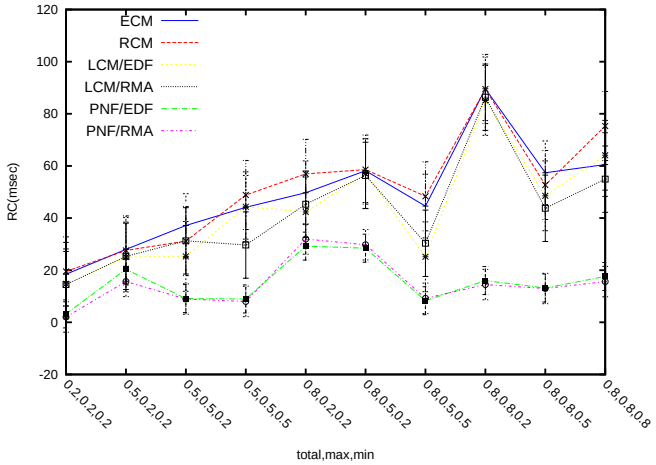


(c) 20 tasks

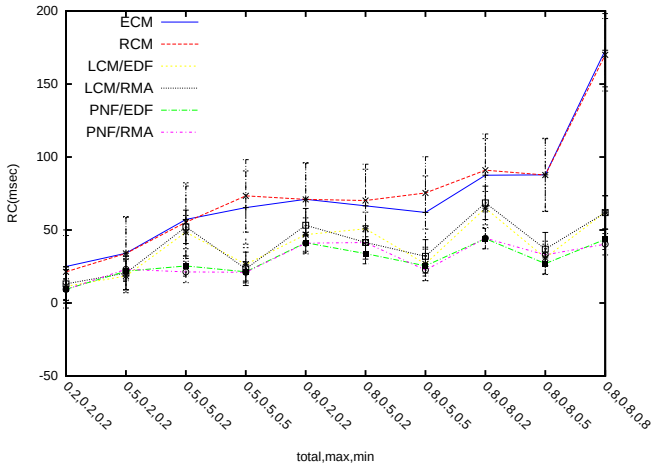
Figure 7.5: Average retry cost for 20 objects per transaction, 100% write operations for different values of total, maximum and minimum atomic section length under different CMs



(a) 4 tasks

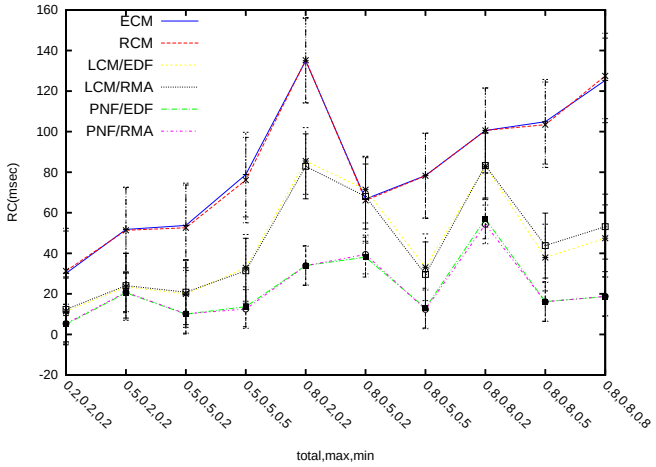


(b) 8 tasks

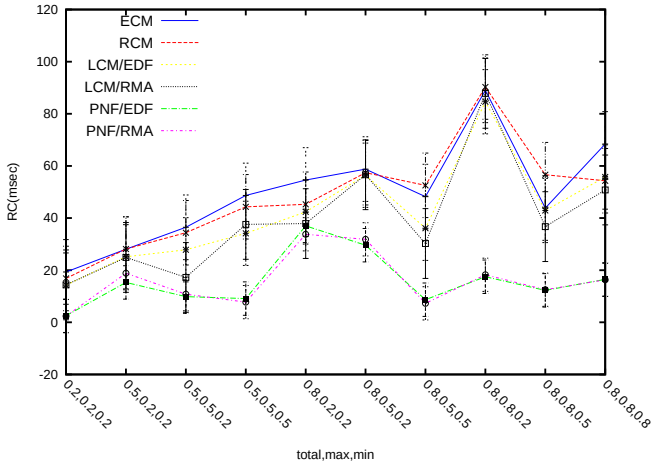


(c) 20 tasks

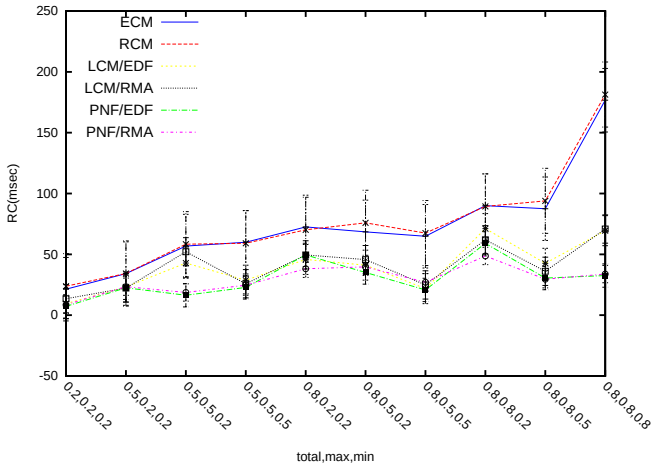
Figure 7.6: Average retry cost for 40 objects per transaction, 40% write operations for different values of total, maximum and minimum atomic section length under different CMs



(a) 4 tasks

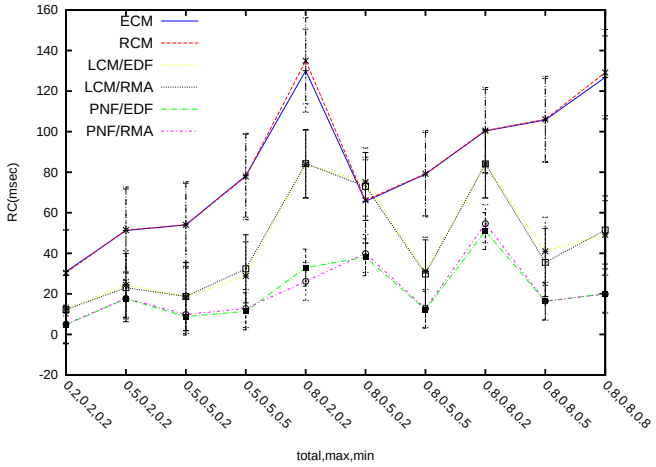


(b) 8 tasks

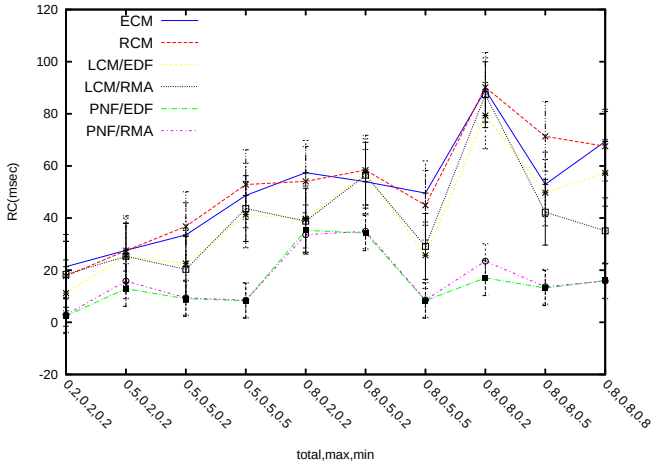


(c) 20 tasks

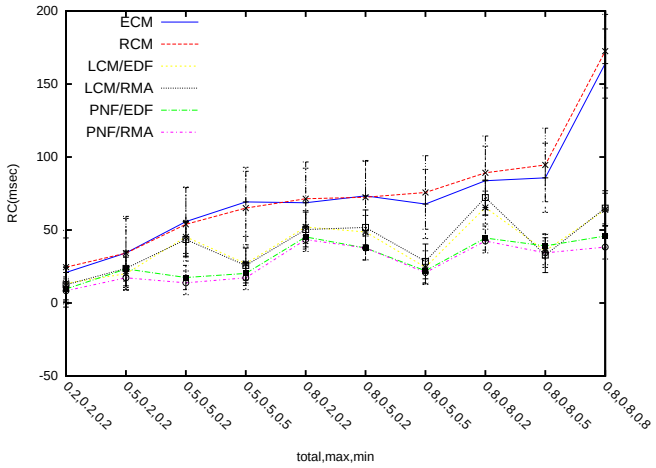
Figure 7.7: Average retry cost for 40 objects per transaction, 80% write operations for different values of total, maximum and minimum atomic section length under different CMs



(a) 4 tasks



(b) 8 tasks



(c) 20 tasks

Figure 7.8: Average retry cost for 40 objects per transaction, 100% write operations for different values of total, maximum and minimum atomic section length under different CMs

Chapter 8

Conclusions and Proposed Post Preliminary Exam Work

8.1 Conclusions

In this dissertation, we designed, analyzed, and experimentally evaluated four real-time CMs. Designing real-time CMs is straightforward. The simplest logic is to use the same rationale as that of the underlying real-time scheduler. This was shown in the design of ECM and RCM. ECM allows the transaction with the earliest absolute deadline (i.e., dynamic priority) to commit first. RCM allows the transaction with the smallest period (i.e., fixed priority) to commit first. We established upper bounds for retry costs and response times under ECM and RCM, and identified the conditions under which they have better schedulability than lock-free synchronization.

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 synchronization 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 synchronization 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.

LCM, which can be used with both G-EDF and G-RMA, tries to compromise between priority of transactions (which is the priority of the underlying tasks), and the remaining ex-

ecution time of the interfered transaction. As the remaining execution time of the interfered transaction decreases, it is not effective to abort the transaction when it can shortly commit. The parameters α and ψ are used to determine whether or not to abort the interfered transaction. α ranges between 0 and 1. When $\alpha \rightarrow 0$, LCM defaults to FCFS. When $\alpha \rightarrow 1$, G-EDF/LCM defaults to ECM, and G-RMA/LCM defaults to RCM. We also derived upper bounds on retry costs and response times under LCM, and compared the schedulability of LCM with ECM, RCM, and lock-free synchronization. Also, we identified the conditions under which LCM performs better than the other synchronization techniques. LCM reduces the retry cost of each atomic section to $(1 + \alpha_{max})s_{max}$, instead of $2.s_{max}$ as in 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 and G-RMA/LCM is better or equal to lock-free synchronization as long as s_{max}/r_{max} does not exceed 0.5. By proper choice of αs, s_{max}/r_{max} can be increased to 2 under G-EDF/LCM, and to larger values under G-RMA/LCM.

ECM, RCM, and LCM are affected by transitive retry. Transitive retry occurs when a transaction accesses multiple objects. It causes a transaction to abort and retry due to another non-conflicting transaction. PNF avoids transitive retry, and and also optimizes the processor usage by reducing the priority of the aborted transaction. This way, other tasks can proceed if they do not conflict with other executing transactions.

We upper bounded PNF's retry cost and response time. We also compared PNF's schedulability to other synchronization techniques. PNF has better schedulability than lock-free synchronization as long as s_{max} does not exceed r_{max}.

We also implemented the CMs and conducted experimental studies. Our experimental studies revealed that the CMs' have shorter retry costs than lock-free synchronization. In particular, PNF has shorter retry costs than others as long as transitive retry and contention exist. However, PNF's implementation is relatively complex.

8.2 Proposed Post Preliminary Exam Research

We propose the following research directions:

8.2.1 Supporting Nested Transactions

Transactions can be nested *linearly*, where each transaction has at most one pending transaction [89]. Nesting can also be done in *parallel*, where transactions execute concurrently within the same parent [113]. Linear nesting can be *1) flat:* If a child transaction aborts, then the parent transaction also aborts. If a child commits, no effect is taken until the parent commits. Modifications made by the child transaction are only visible to the parent until the parent commits, after which they are externally visible. *2) Closed:* Similar to *flat nesting*, except that if a child transaction conflicts, it is aborted and retried, without aborting the parent, potentially improving concurrency over flat nesting. *3) Open:* If a child transaction commits, its modifications are immediately externally visible, releasing memory isolation of objects used by the child, thereby potentially improving concurrency over closed nesting. However, if the parent conflicts after the child commits, then compensating actions are executed to undo the actions of the child, before retrying the parent and the child.

We propose to develop real-time contention managers that allow these different nesting models and establish their retry and response time upper bounds. Additionally, we propose to formally compare their schedulability with nested critical sections under lock-based synchronization. Note that, nesting is not viable under lock-free synchronization.

The real-time CMs proposed so far can be directly extended for different types of nesting. On conflict, the CM should decide on which transaction must be aborted. In case of flat nesting, the outer transaction should be aborted and restarted. In case of closed and open nesting, only the interfered transaction should be restarted. In flat nesting, it will be useful to delay a lower priority transaction when it interferes with a higher priority one. Thus, the lower priority transaction need not be restarted from the beginning. This can reduce retry costs, especially when the inner most transaction is the conflicting one.

ECM and RCM can use the same criteria to determine which transaction to abort or wait. LCM may need a redefinition of the α parameter. Each child transaction can have its own α. So, for each depth of nesting, there is a corresponding α. Alternatively, there can be only one α for the outer transaction. Inner transactions do not have their αs. The first choice may be suitable for closed and open nesting, while the second choice seems more suitable for flat nesting.

PNF can be used directly with all types of nesting. The only requirement is that each transaction checks for conflicts with itself, as well as its inner transactions. Executing transactions under PNF are non-preemptive. Thus, any nested transaction will not be aborted. This requirement for PNF simplifies implementation, but makes no use of the nesting structure (i.e., PNF deals with transactions as if they are not nested). The previous requirement for PNF can be alleviated by checking for conflicts of any inner transaction only when the inner transaction begins. So, when a transaction starts, it checks its own objects – not inner transactions – against objects of executing transactions. If no conflict is found, the transaction executes. Consequently, when an inner transaction starts, it can detect a conflict

with already executing transactions. Thus, the inner transaction should wait until other conflicting executing transactions commit. This choice for the PNF implementation may add additional waiting time to inner transactions. Different choices thus impose different tradeoffs.

Worst case response time analysis, similar to the design of CMs, needs modifications to cope with nested transactions. The simplest method is to combine all accessed objects in all nested transactions into one group, and consider that this group is accessed by only one non-nested transaction. This simplifies calculations but loosens the upper bounds, especially with closed and open nesting. Since closed and open nesting only restarts the inner aborted transaction, it is of no use to include the length of the outermost parent in the response time calculations.

8.2.2 Combining and Optimizing LCM and PNF

LCM is designed to reduce the retry cost of a transaction when it is interfered close to the end of its execution. In contrast, PNF is designed to avoid transitive retry when transactions access multiple objects. An interesting direction is to combine the two contention managers to obtain the benefits of both algorithms.

An important problem in developing such a LCM/PNF combination is that LCM allows aborts of running transactions depending on the α value, whereas PNF does not permit aborts of any executing transaction. Thus, the first approach to combine LCM and PNF is by considering the transitive retry level. If the transitive retry level is high, then PNF could be used. Otherwise, LCM could be used. Another way to combine LCM and PNF is to change the α value with time. Thus, at some time point, α can equal 0. This means that the current transaction cannot be aborted by any other transaction. This is similar to executing transactions in PNF. However, α should not be 0 all the time. Thus, α should change with time.

For such a combined LCM/PNF contention manager, importantly, we must understand what are its schedulability advantages over that of LCM and PNF individually, and how such a combined CM behaves in practice.

The current implementation of PNF is centralized. In this implementation, PNF acts as a centralized contention manager that controls all transactions. Contention managers are usually decentralized, as described in Section 2.4. Each transaction usually maintains its own contention manager; this reduces overhead and transaction blocking. Thus, one optimization direction is to develop a decentralized implementation of PNF, to reduce overhead and blocking. Moreover, the current implementation of PNF uses locks, which increases overhead. Another optimization direction is therefore to develop a lock-free PNF implementation.

Design optimizations of LCM and PNF may also be possible to reduce their retry costs and response times, by considering additional criteria for resolving transactional conflicts.

Developing a LCM/PNF combination and optimizing PNF and LCM implementations and designs constitute our second research direction.

8.2.3 Formal and Experimental Comparison with Real-Time Locking

Lock-free synchronization offers numerous advantages over locking protocols, but (coarse-grain) locking protocols have had significant traction in real-time systems due to their good programmability (even though their concurrency is low). Example such real-time locking protocols include PCP and its variants [26, 69, 92, 102], multicore PCP (MPCP) [73, 91], SRP [8,23], multicore SRP (MSRP) [49], PIP [38], FMLP [16,17,64], and OMLP [11]. FMLP has been established to be superior to other protocols [20]. How does their schedulability compare with that of the proposed contention managers? How do they compare in practice? These questions constitute our third research direction.

OMLP [18] is similar to FMLP, and is simpler in implementation. OMLP does not require changes in the underlying scheduler (i.e., G-EDF) as FMLP does. Under OMLP, each resource has a FIFO queue of length at most equals number of processors, m, and a priority queue. Requests for each resource are enqueued in the corresponding FIFO queue. If the FIFO queue is full, requests are added to the priority queue according to the requesting job's priority. The head of the FIFO queue is the resource holding task. Other queued requests are suspended until their turn arrives. Under suspension oblivious analysis for OMLP, each job has a maximum blocking time that is proportional to number of processors. In suspension oblivious, the suspension time is added to task execution, in contrast to suspension aware analysis. This is why OMLP is asymptotically optimal under suspension oblivious analysis. We intend to implement OMLP in the ChronOS real-time OS. We also propose to analytically and experimentally compare OMLP against different CMs, including that for nested and non-nested transactions.

Bibliography

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 316 – 327, feb. 2005.

[2] 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.

[3] J.H. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *17th IEEE Real-Time Systems Symposium*, pages 94 –105, dec 1996.

[4] J.H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 28 –37, dec 1995.

[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] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management asanon-clairvoyant scheduling problem. *Algorithmica*, 57:44–61, 2010. 10.1007/s00453-008-9195-x.

[7] Tian Bai, YunSheng Liu, and Yong Hu. Timestamp vector based optimistic concurrency control protocol for real-time databases. In *4th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM)*, pages 1 –4, oct. 2008.

[8] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3:67–99, 1991.

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

[10] A. Barros and L.M. Pinho. Software transactional memory as a building block for parallel embedded real-time systems. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 251 –255, 30 2011-sept. 2 2011.

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

[12] M. Behnam, F. Nemati, T. Nolte, and H. Grahn. Towards an efficient approach for resource sharing in real-time multiprocessor systems. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 99 –102, june 2011.

[13] C. Belwal and A.M.K. Cheng. Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *IEEE Embedded Systems Letters*, 3(1):37 –41, march 2011.

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

[15] 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.

[16] 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)*, pages 47 –56, aug. 2007.

[17] 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.

[18] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 49 –60, 30 2010-dec. 3 2010.

[19] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 69 –78, oct. 2011.

[20] 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, 2008.

[21] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In Antonio Fernndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 207–221. Springer Berlin-Heidelberg, 2011.

[22] G.C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer-Verlag New York Inc, 2005.

[23] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[24] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[25] 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.

[26] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990.

[27] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. Synchronization for an optimal real-time scheduling algorithm on multiprocessors. In *International Symposium on Industrial Embedded Systems (SIES)*, pages 9 –16, july 2007.

[28] 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.

[29] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 101 –110, dec. 2006.

[30] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. Lock-free synchronization for dynamic embedded real-time systems. *ACM Trans. Embed. Comput. Syst.*, 9(3):23:1–23:28, March 2010.

[31] 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.

[32] D. Christie, J.W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[33] 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.

[34] A. Datta, S.H. Son, and V. Kumar. Is a bird in the hand worth more than two in the bush? limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482 –502, may 2000.

[35] 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.

[36] U.M.C. Devi, H. Leontyev, and J.H. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *18th Euromicro Conference on Real-Time Systems*, pages 10 pp. –84, 0-0 2006.

[37] 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.

[38] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377 –386, dec. 2009.

[39] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *19th RTNS*, 2011.

[40] M. Elshambakey and B. Ravindran. On real-time stm concurrency control with improved schedulability. Submitted to *EMSOFT'12*.

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

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

[43] J.R. Engdahl and Dukki Chung. Lock-free data structure for multi-core processors. In *International Conference on Control Automation and Systems (ICCAS)*, pages 984 –989, oct. 2010.

[44] A. Ermedahl, H. Hansson, M. Papatriantafilou, and P. Tsigas. Wait-free snapshots in real-time systems: algorithms and performance. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 257 –266, oct 1998.

[45] 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.

[46] 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.

[47] Y.M.P. Fernandes, A. Perkusich, P.F.R. Neto, and M.L.B. Perkusich. Implementation of transactions scheduling for real-time database management. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 6, pages 5136 – 5141 vol.6, oct. 2004.

[48] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[49] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189 – 198, may 2003.

[50] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 73 – 83, dec. 2001.

[51] J. Gottschlich and D.A. Connors. Extending contention managers for user-defined priority-based transactions. In *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM), Boston, MA*. Citeseer, 2008.

[52] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323. 2005.

[53] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[54] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Towards a theory of transactional contention managers. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, pages 316–317, New York, NY, USA, 2006. ACM.

[55] 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.

[56] M. Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. In *Unpublished Manuscript*. Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[57] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

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

[59] 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.

[60] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In Gadi Taubenfeld, editor, *Distributed Computing*, volume 5218 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin / Heidelberg, 2008.

[61] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 82–91, New York, NY, USA, 2006. ACM.

[62] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, 2001.

[63] P. Holman and J.H. Anderson. Locking in pfair-scheduled multiprocessor systems. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 149 – 158, 2002.

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

[65] P. Holman and J.H. Anderson. Supporting lock-free synchronization in Pfair-scheduled real-time systems. *Journal of Parallel and Distributed Computing*, 66(1):47–67, 2006.

[66] Philip L. Holman. *On the implementation of pfair-scheduled multiprocessor systems*. PhD thesis, University of North Carolina, Chapel Hill, 2004.

[67] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf, 2007.

[68] Intel Corporation. Intel Itanium Architecture Software Developers Manual Volume 3: Instruction Set Reference. http://download.intel.com/design/Itanium/manuals/24531905.pdf, 2007.

[69] 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.

[70] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[71] Tei-Wei Kuo and Hsin-Chia Hsih. Concurrency control in a multiprocessor real-time database system. In *12th Euromicro Conference on Real-Time Systems (Euromicro RTS)*, pages 55 –62, 2000.

[72] Shouwen Lai, Binoy Ravindran, and Hyeonjoong Cho. On scheduling soft real-time tasks with lock-free synchronization for embedded devices. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1685–1686, New York, NY, USA, 2009. ACM.

[73] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 469 –478, dec. 2009.

[74] Kam-Yiu Lam, Tei-Wei Kuo, and Wai-Hung Tsang. Concurrency control for real-time database systems with mixed transactions. In *Proceedings of Fourth International Workshop on Real-Time Computing Systems and Applications*, pages 96 –103, oct 1997.

[75] C.P.M. Lau and V.C.S. Lee. Real time concurrency control for data intensive applications. In *Proceedings of 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 337 – 342, aug. 2005.

[76] M.R. Lehr, Young-Kuk Kim, and S.H. Son. Managing contention and timing constraints in a real-time database system. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 332 –341, dec 1995.

[77] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 197–206, New York, NY, USA, 2008. ACM.

[78] Gertrude Levine. Priority inversion with fungible resources. *Ada Lett.*, 31(2):9–14, February 2012.

[79] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[80] G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262 –271, july 2011.

[81] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Riviere. Deadline-aware scheduling for software transactional memory. In *41st International Conference on Dependable Systems Networks (DSN)*, pages 257 –268, june 2011.

[82] 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.

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

[84] 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.

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

[86] Fadi Meawad, Martin Schoeberl, Karthik Iyer, and Jan Vitek. Real-time wait-free queues using micro-transactions. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 1–10, New York, NY, USA, 2011. ACM.

[87] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[88] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, feb. 2006.

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

[90] Wu Peng and Pang Zilong. Research on the improvement of the concurrency control protocol for real-time transactions. In *International Conference on Machine Vision and Human-Machine Interface (MVHI)*, pages 146 –148, april 2010.

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

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

[93] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of 32nd International Symposium on Computer Architecture (ISCA)*, pages 494 – 505, june 2005.

[94] M. Raynal. Wait-free objects for real-time systems? In *Proceedings of Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 413 –420, 2002.

[95] T. Riegel, P. Felber, and C. Fetzer. TinySTM. http://tmware.org/tinystm, 2010.

[96] 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.

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

[98] 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.

[99] W.N. Scherer III and M.L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

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

[101] M. Schoeberl and P. Hilber. Design and implementation of real-time transactional memory. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 279 –284, 31 2010-sept. 2 2010.

[102] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175 –1185, sep 1990.

[103] L. Sha, R. Rajkumar, S.H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793 –800, jul 1991.

[104] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[105] 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.

[106] Richard L. Sites. Alpha AXP architecture. *Commun. ACM*, 36:33–44, February 1993.

[107] 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.

[108] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58 –71, nov 1993.

[109] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications*, pages 433 –440, 2000.

[110] 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.

[111] P. Tsigas, Yi Zhang, D. Cederman, and T. Dellsen. Wait-free queue algorithms for the real-time java specification. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 373 – 383, april 2006.

[112] University of Rochester. Rochester Software Transactional Memory. http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml, http://code.google.com/p/rstm, 2006.

[113] 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.

[114] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261 –272, feb. 2007.

[115] Kam yiu Lam, Tei-Wei Kuo, and T.S.H. Lee. Designing inter-class concurrency control strategies for real-time database systems with mixed transactions. In *12th Euromicro Conference on Real-Time Systems (Euromicro RTS)*, pages 47 –54, 2000.

[116] P.S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and S.H. Son. On real-time databases: concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140 –157, jan 1994.