On Real-Time STM Concurrency Control for Embedded Software with Improved Schedulability

Mohammed Elshambakey
ECE Dept.
Blacksburg, VA 24060, USA
shambake@vt.edu

Binoy Ravindran
ECE Dept.
Blacksburg, VA 24060, USA
binoy@vt.edu

Abstract— We consider software transactional memory (STM) concurrency control for embedded multicore real-time software, and present a novel contention manager for resolving transactional conflicts, called PNF. We upper bound transactional retries and task response times. Our implementation in RSTM/real-time Linux reveals that PNF yields shorter or comparable retry costs than competitors.

I. Introduction

Concurrency is intrinsic to embedded software, as they control concurrent physical processes. Often, such concurrent computations need to read/write shared data objects. They must also satisfy time constraints.

Lock-based concurrency control has significant programmability, scalability, and composability challenges [9]. Software transactional memory (STM) is an alternative synchronization model for shared memory objects that promises to alleviate these difficulties. With STM, code that read/write shared objects is organized as transactions, which execute speculatively, while logging changes made to objects. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager (CM) [7] 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. Besides a simple programming model, STM provides performance comparable to lock-free synchronization, especially for read-dominated workloads, and is composable [8].

Given STM's programmability, scalability, and composability advantages, it is a compelling concurrency control technique also for multicore embedded real-time software. However, this requires bounding transactional retries, as real-time threads, which subsume transactions, must satisfy time constraints. STM retry bounds are dependent on the CM policy.

Past real-time CM research has proposed resolving transactional contention using dynamic and fixed priorities of parent threads, resulting in Earliest Deadline First CM (ECM) and Rate Monotonic CM (RCM), respectively [6, 5, 4]. In particular, [5] shows that ECM and RCM achieve higher schedulability – i.e., greater number of task sets meeting their time constraints – than lock-free synchronization only under some ranges for the maximum atomic section length. That range is significantly expanded with the Length-based CM (LCM) in [4], increasing the coverage of STM's timeliness superiority. However, these works restrict to *one* object access per transaction, which is a major limitation (Section III). To allow multiple objects per transaction, we design a novel contention manager called PNF (Section IV), which can be used with global EDF (G-EDF) and global RMA (G-RMA) multicore real-time schedulers. We upper bound transactional retry costs and task response times under PNF (Section V). Our implementation reveals that PNF yields shorter or comparable retry costs than competitors (Section VI).

PNF's superior timeliness properties thus allow embedded real-time programmers to reap STM's significant programmability and composability advantages for a broader range of multicore embedded real-time software than what was previously possible – paper's contribution.

II. Preliminaries

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

Shared objects. A task may need to read/write shared, in-memory data objects while it is executing any of its atomic sections (transactions), 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. $p(s_i^k)$ is the priority of transaction s_i^k. Each object, θ, can be accessed by multiple tasks. The set of distinct objects accessed by τ_i is θ_i without repeating objects. 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 can access one or more objects in θ_i. So, s_i^k refers to the transaction itself, regardless of the objects accessed by the transaction. We denote the set of all accessed objects by s_i^k as Θ_i^k. While $s_i^k(\theta)$ implies that s_i^k accesses an object $\theta \in \Theta_i^k$, $s_i^k(\Theta)$ implies that s_i^k accesses a set of objects $\Theta = \{\theta : \theta \in \Theta_i^k\}$. $\bar{s_i^k} = \bar{s_i^k}(\Theta)$ refers only once to s_i^k, regardless of the number of objects in Θ. So, $|\bar{s_i^k}(\Theta)|_{\forall \theta \in \Theta} = 1$.

$s_i^k(\theta)$ executes for a duration $len(s_i^k(\theta))$. $len(s_i^k) = len(s_i^k(\theta)) = len(s_i^k(\Theta)) = len(s_i^k(\Theta_i^k))$. The set of tasks sharing θ with τ_i is denoted $\gamma_i(\theta)$. Atomic sections are non-nested (supporting nested STM 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 $s^i_{max}(\theta)$.

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 a conflict with another section $s_j^k(\theta)$, is denoted $W_i^p(s_j^k(\theta))$. If an atomic section, s_i^p, is already executing, and another atomic section s_j^k tries to access a shared object with s_i^p, then s_j^k is said to "interfere" or "conflict" with s_i^p. The transaction s_j^k is the "interfering transaction", and the transaction s_i^p is the "interfered transaction".

Due to *transitive retry* (Definition 1) an atomic section $s_i^k(\Theta_i^k)$ may retry due to another atomic section $s_j^l(\Theta_j^l)$, where $\Theta_i^k \cap \Theta_j^l = \emptyset$. θ_i^* denotes the set of objects not accessed directly by atomic sections in τ_i, but can cause transactions in τ_i to retry due to transitive retry. $\theta_i^{ex}(= \theta_i + \theta_i^*)$ is the set of all objects that can cause transactions in τ_i to retry directly or through transitive retry. γ_i^* is the set of tasks that accesses objects in θ_i^*. $\gamma_i^{ex}(= \gamma_i + \gamma_i^*)$ is the set of all tasks that can directly or indirectly (through transitive retry) cause transactions in τ_i to retry.

The total time that a task τ_i's atomic sections have to retry over T_i is denoted $RC(T_i)$. The additional amount of time by which all interfering jobs of τ_j increases the response time of any job of τ_i during L, without considering retries due to atomic sections, is denoted $W_{ij}(L)$.

III. Limitations of ECM, RCM, and LCM

ECM and RCM [5] use dynamic and fixed priorities, respectively, to resolve conflicts. ECM uses G-EDF, and allows the transaction whose job has the earliest absolute deadline to commit first [6]. RCM uses G-RMA, and commits the transaction whose job has the shortest period. To use STM in real-time systems, transactional retry cost must be bounded in order to satisfy time constraints. ECM's retry cost is bounded in [5] as follows:

Claim 1 (from [5]): Under ECM, a task τ_i's maximum retry cost during T_i is upper bounded by:

$$RC\left(T_i\right) \leq \sum_{\theta\in\theta_i}\Bigg(\Big(\sum_{\tau_j\in\gamma_i(\theta)}\Big(\left\lceil\frac{T_i}{T_j}\right\rceil\sum_{\forall s_j^l(\theta)} len\big(s_j^l(\theta) + s_{max}(\theta)\big)\Big)\Big) - s_{max}(\theta) + s_{i_{max}}(\theta)\Bigg) \quad (1)$$

Retry cost under RCM is similar to ECM, except that, only tasks with higher priority than τ_i can interfere with any job τ_i^x of τ_i. RCM's retry cost is also bounded in [5].

G-EDF/LCM [4] and G-RMA/LCM default to ECM and RCM, respectively, with some difference. Under LCM, a higher priority transaction $s_i^k(\theta)$ cannot abort a lower priority transaction $s_j^l(\theta)$ if $s_j^l(\theta)$ has already consumed α percentage of its execution length. G-EDF/LCM's retry cost is bounded in [4] as follows:

Claim 5 (from [4]): $RC(T_i)$ for a task τ_i under G-EDF/LCM is upper bounded by:

$$\begin{aligned} RC(T_i) &= \Bigg(\sum_{\forall\tau_h\in\gamma_i}\sum_{\forall\theta\in\theta_i\wedge\theta_h}\Bigg(\left\lceil\frac{T_i}{T_h}\right\rceil\sum_{\forall s_h^l(\theta)} len\Big(s_h^l(\theta)\Big) \\ &+ \alpha_{max}^{hl} len\Big(s_{max}^h(\theta)\Big)\Bigg)\Bigg) \\ &+ \sum_{\forall s_i^y(\theta)}\Big(1-\alpha_{max}^{iy}\Big) len\Big(s_{max}^i(\theta)\Big) \end{aligned} \quad (2)$$

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

G-RMA/LCM's retry cost is similar to G-EDF/LCM's, except that, only tasks with higher priority than τ_i can interfere with any job τ_i^x of τ_i. G-RMA/LCM's retry cost is bounded in [4].

As mentioned before, [5, 4] assumes that each transaction accesses only one object. This assumption simplifies the retry cost and response time analysis [5, 4]. Besides, it enables comparison with lock-free synchronization [3]. 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 s_i^k retries due to a higher priority transaction s_z^h, and $\Theta_z^h \cap \Theta_i^k = \emptyset$.*

Claim 1 *ECM, RCM and LCM suffer from transitive retry for multi-object transactions.*

Proof 1 Example 1 applies for any transaction under ECM, RCM, and LCM. Claim follows.

Hence, the analysis in [5, 4] 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 if any exists.

Algorithm 1: PNF Algorithm

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

1 **if** s_i^k *does not conflict with any executing transaction* **then**
2   Assign s_i^k as an executing transaction;
3   Add s_i^k to the m-set;
4   Select s_i^k to commit
5 **else**
6   Add s_i^k to the n-set according to its priority;
7   Assign temporary priority -1 to the job that owns s_i^k ;
8   Select transaction(s) conflicting with s_i^k for commit;
9 **end**
10 **if** s_j^l *commits* **then**
11   **for** *z=1 to size of n-set* **do**
12   **if** *n(z) does not conflict with any executing transaction* **then**
13   **if** *processor available*[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   **end**
23 **end**

IV. The PNF Contention Manager

Algorithm 1 describes PNF. It manages two sets. The first is the m-set, which contains at most m non-conflicting transactions, where m is the number of processors, as there cannot be more than m executing transactions (or generally, m executing jobs) at the same time. When a transaction is entered in the m-set, it executes non-preemptively and no other transaction can abort it. A transaction in the m-set is called an *executing transaction*. This means that, when a transaction is executing before the arrival of higher priority conflicting transactions, then the one that started executing first will be committed (Step 8) (hence the term "First access" in PNF).

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 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 PNF). 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

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

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

the original priority of transactions (hence the term "Priority" 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.

A. Properties

Claim 2 *Transactions scheduled under PNF do not suffer from transitive retry.*

Proof 2 Proof is by contradiction. Assume three transactions s_i^k, s_j^l, and s_z^h. $p(s_z^h) > p(s_j^l) > p(s_i^k)$. $\Theta_i^k \cap \Theta_j^l \neq \emptyset$, $\Theta_j^l \cap \Theta_z^h \neq \emptyset$, but $\Theta_i^k \cap \Theta_z^h = \emptyset$. Assume s_i^k is transitively retrying because of s_z^h. This means, s_i^k, s_j^l, and s_z^h are executing concurrently. $\Theta_i^k \cap \Theta_j^l \neq \emptyset$, so s_i^k and s_j^l cannot be executing transactions at the same time, by the definition of PNF. $\Theta_j^l \cap \Theta_z^h \neq \emptyset$, so s_j^l and s_z^h cannot be executing transactions at the same time, by the definition of PNF. Only s_i^k and s_z^h can be executing transactions at the same time, because $\Theta_i^k \cap \Theta_z^h = \emptyset$. Hence, the three transactions cannot be running concurrently. So, s_i^k cannot be transitively retrying because of s_z^h, which contradicts the first assumption. Claim follows.

From Claim 2, 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 1.

Claim 3 *Under PNF, any job τ_i^x is not affected by the retry cost in any other job τ_j^l.*

Proof 3 As explained in Section 1, 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.

V. 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 4 *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 4 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 5 *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:*

$$RC(L) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^l}(\theta)} len\left(\bar{s_j^l}(\theta) \right) \right) \right) \tag{3}$$

Proof 5 Consider a transaction s_i^k belonging to job τ_i^x. By definition of PNF, higher and lower priority transactions than s_i^k can become executing transaction before s_i^k. 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. Executing transactions are not aborted. This is why $\bar{s_j^l}$ is included only once in (3) 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 4, Claim follows.

Claim 6 *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:*

$$D(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h} \right) \right) \right\rfloor \tag{4}$$

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 $\ddot{s_j^h} = \{s_j^h : (\Theta_j^h \cap \Theta_i^k = \emptyset) \wedge (\forall \Theta_i^k \in \theta_i)\}$. During this blocking time, all processors are unavailable for τ_i^x.

Proof 6 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. If lower priority transactions are conflicting with any transaction in τ_i^x, then (3) already covers the increase in retry cost of transactions in τ_i^x due to lower priority transactions. Otherwise, lower priority non-conflicting transactions can be executing transactions that block τ_i^x.

Lower priority non-conflicting transactions can block τ_i^x when τ_i^x is newly released, or after that:

Lower priority non-conflicting transactions when τ_i^x is newly released. τ_i^x is delayed if no processor is available. 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 transactions and/or higher priority jobs. Otherwise, τ_i^x can run in parallel with 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 [1].

Claim 7 *A job τ_i^x's response time, during an interval $R_i^{up} \leq T_i$, under PNF/G-EDF is upper bounded by:*

$$R_i^{up} = c_i + RC(R_i^{up}) + D_{edf}(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} W_{ij}(R_i^{up}) \right\rfloor \quad (5)$$

where $RC(R_i^{up})$ is calculated by (3). $D_{edf}(\tau_i^x)$ is the same as $D(\tau_i^x)$ defined in (4). However, for G-EDF, $D_{edf}(\tau_i^x)$ is calculated as:

$$D_{edf}(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \begin{cases} 0 & , R_i^{up} \leq T_i - T_j \\ \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h}\right) & , R_i^{up} > T_i - T_j \end{cases} \right\rfloor \quad (6)$$

and $W_{ij}(R_i^{up})$ is calculated by (3) in [5].

Proof 7 τ_i^x's response time is calculated by (3) in [5] with the addition of blocking time defined by Claim 6. 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 $R_i^{up} < T_i - T_j$, no existing instance of τ_j is of lower priority than τ_i^x. Hence, 0 is used in the first case of (6). But if $R_i^{up} > T_i - T_j$, there can be only one instance τ_j^h of τ_j with lower priority than τ_i^x. Hence, $\left\lceil \frac{R_i^{up}}{T_i} \right\rceil + 1$ in (4) is replaced with 1 in the second case in (6). Claim follows.

Claim 8 *A job τ_i^x's response time, during an interval $R_i^{up} \leq T_i$, under PNF/G-RMA is upper bounded by:*

$$R_i^{up} = c_i + RC(R_i^{up}) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i, p_j > p_i} W_{ij}(R_i^{up}) \right\rfloor \quad (7)$$

where $RC(L)$ is calculated by (3), $D(\tau_i^x)$ is calculated by (4), and $W_{ij}(R_i^{up})$ is calculated by (2) in [5].

Proof 8 Proof is same as of Claim 7, except that G-RMA assigns fixed priorities. Hence, (4) can be used directly for calculating $D(\tau_i^x)$ without modifications. Claim follows.

VI. Experimental Evaluation

To understand how PNF's retry cost compares with competitors in practice (i.e., on average), we implement PNF and the competitors and conduct experiments.

We used the ChronOS real-time Linux kernel [2] and the RSTM library [10] in our implementation. We implemented G-EDF and G-RMA schedulers in ChronOS, and modified RSTM to include implementations of ECM, RCM, LCM, and PNF. For the retry-loop lock-free synchronization, we used a loop that reads an object and attempts to write to it using a CAS instruction. The task retries until the CAS succeeds. We used an 8 core, 2GHz AMD Opteron platform. The average time taken for one write operation by RSTM on any core is 0.0129653375μs, and the average time taken by one CAS-loop operation on any core is 0.0292546250 μs.

We used four task sets consisting of 4, 5, 8, and 20 periodic tasks. Each task runs in its own thread and has a set of atomic sections. Atomic section properties are probabilistically controlled using: 1) the maximum and 2) minimum lengths of any atomic section within a task, and 3) the total length of atomic sections within any task. Since lock-free synchronization cannot handle more than one object per atomic section, we first compare PNF's retry cost with that of lock-free (and other CMs) for one object per transaction. We then compare PNF's retry cost with that of other CMs for multiple objects per transaction.

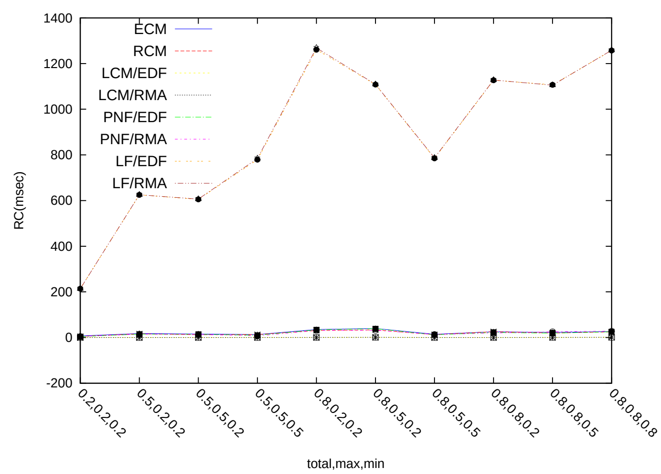

Fig. 1. Avg. retry cost (1 shared object, 5 tasks).

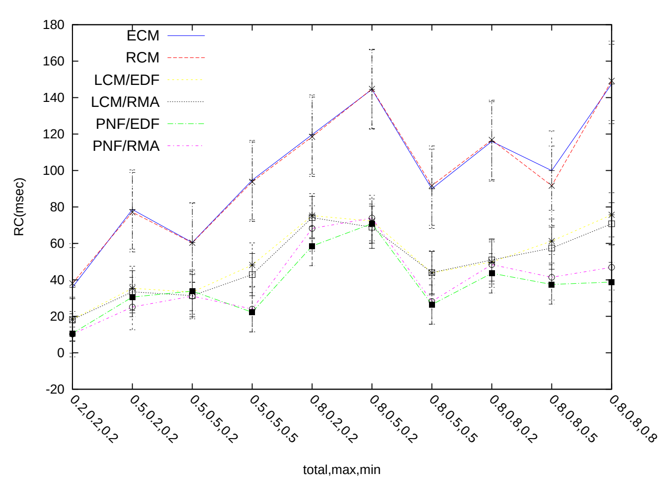

Fig. 2. Avg. retry cost (5 shared objects, 4 tasks).

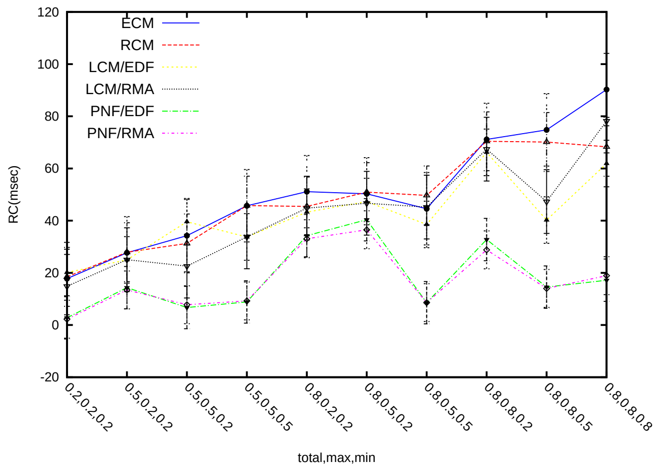

Fig. 3. Avg. retry cost (20 shared objects, 8 tasks).

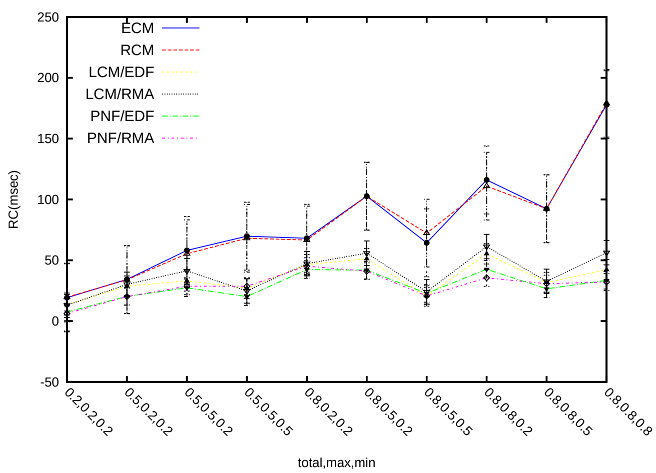

Fig. 4. Avg. retry cost (20 shared objects, 20 tasks).

Figures 1, 2, 3, 4 and 5 show the average retry cost for the 5, 4, 8 and 20 tasks, under 1, 5, 20 and 40 shared objects, respectively. On the x-axis of the figure, we record 3 parameters x, y, and z. x is the ratio of the total length of all atomic sections of a task to the task WCET. y is the ratio of the maximum length of any atomic section of a task to the task WCET. z is the ratio of the minimum length of any atomic section of a task to the task WCET. Confidence level of all data points is 0.95.

Figure 1 compares ECM, RCM, LCM, PNF and lock-free with one shared object per transaction. Lock-free has the largest retry cost, as it provides no conflict resolution. PNF's retry cost closely approximates ECM's and RCMs, as there is no transitive retry because there is only one shared object. Figures 2, 3, 4 and 4 compare CMs under shared multiple objects per transaction. We observe that PNF has shorter or comparable retry cost than ECM, RCM, and LCM. Similar trends were observed for the other task sets; those are omitted here for brevity and due to space limitations.

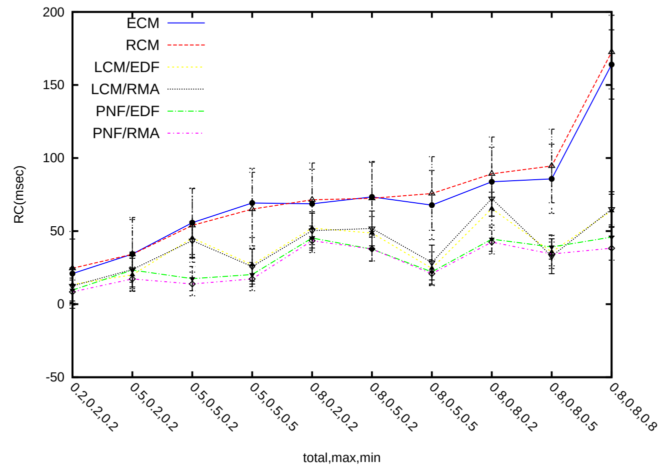

Fig. 5. Avg. retry cost (40 shared objects, 20 tasks).

VII. Conclusions

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

Acknowledgment

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

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

[2] M. Dellinger et al. ChronOS Linux: a best-effort real-time multiprocessor Linux kernel. In *DAC*, pages 474–479, 2011.

[3] U. Devi et al. Efficient synchronization under global EDF scheduling on multiprocessors. In *ECRTS*, pages 10 pp. –84, 2006.

[4] M. El-Shambakey and B. Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *DAC*, pages 437–446, 2012.

[5] M. El-Shambakey and B. Ravindran. STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs. In *SAC*, pages 1602–1609, 2012.

[6] S. Fahmy and B. Ravindran. On STM concurrency control for multicore embedded real-time software. In *SAMOS*, pages 1 –8, 2011.

[7] R. Guerraoui et al. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[8] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51:91–100, Aug 2008.

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

[10] V. J. Marathe et al. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, 2006.