FBLT: A Real-Time Contention Manager with Improved Schedulability

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

Binoy Ravindran
ECE Dept, Virginia Tech
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 FBLT. We upper bound transactional retries and task response times under FBLT, and identify when FBLT has better real-time schedulability than the previous best contention manager, PNF. Our implementation in the Rochester STM framework reveals that FBLT yields shorter or comparable retry costs than competitor methods.**

I. 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 while satisfying time constraints.

The de facto standard for concurrent programming is the threads abstraction, and the de facto synchronization abstraction is locks. Lock-based concurrency control has significant programmability, scalability, and composability challenges [1]. Transactional memory (TM) is an alternative synchronization model for shared memory objects that promises to alleviate these difficulties. With TM, code that read/write shared objects is organized as *memory transactions*, which execute speculatively, while logging changes made to objects. Two transactions conflict if they access the same object and at least one access is a write. When that happens, a contention manager (CM) [2] resolves the conflict by aborting one and allowing the other to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling back the changes. In addition to a simple programming model, TM provides performance comparable to lock-free approach, especially for high contention and read-dominated workloads (see an example TM system's performance in [3]), and is composable [4]. TM has been proposed in hardware, called HTM, and in software, called STM, with the usual tradeoffs: HTM has lesser overhead, but needs transactional support in hardware; STM is available on any hardware.

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. Retry bounds under STM are dependent on the CM policy at hand.

Past real-time CM research has proposed resolving transactional contention using dynamic and fixed priorities of parent threads, resulting in Earliest Deadline CM (ECM) and Rate Monotonic CM (RCM) [5]–[7], which are intended to be used with global EDF (G-EDF) and global RMS (G-RMS) multicore real-time schedulers [8], respectively. In particular, [6] 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 [7], increasing the coverage of STM's timeliness superiority. ECM, RCM, and LCM suffer from transitive retry (Section III) and cannot handle multiple objects per transaction efficiently. These limitations are overcome with the Priority with Negative value and First access CM (PNF) [9], [10]. However, PNF requires a-priori knowledge of all objects accessed by each transaction. This significantly limits programmability, and is incompatible with dynamic STM implementations [11]. Additionally, PNF is a centralized CM, which increases overheads and retry costs, and has a complex implementation.

We propose the First Bounded, Last Timestamp CM (or FBLT) (Section IV). In contrast to PNF, FBLT does not require a-priori knowledge of objects accessed by transactions. Moreover, FBLT allows each transaction to access multiple objects with shorter transitive retry cost than ECM, RCM and LCM. Additionally, FBLT is a decentralized CM and does not use locks in its implementation. Implementation of FBLT is also simpler than PNF. We establish FBLT's retry and response time upper bounds under G-EDF and G-RMA schedulers (Section V). We also identify the conditions under which FBLT's schedulability is better than PNF (Section VI). We implement FBLT and competitor CM techniques in the Rochester STM framework [12] and conduct experimental studies (Section VII). Our results reveal that FBLT has shorter retry cost than ECM, RCM, LCM and lock-free. FBLT's retry cost is comparable to that of PNF, especially in case of non-transitive retry, but it doesn't require a-priori knowledge of objects accessed by transactions, unlike PNF.

Thus, the paper's contribution is the FBLT contention manager with superior timeliness properties. FBLT, thus allows programmers to reap STM's significant programmability and composability benefits for a broader range of multicore embedded real-time software than what was previously possible.

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. 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 $\Theta = \{\theta \in \Theta_i^k\}$. $s_i^k = s_i^k(\Theta)$ refers only once to s_i^k, regardless of the number of objects in Θ. So, $|s_i^k(\Theta)|_{\forall\theta\in\Theta} = 1$. $s_i^k(\theta)$ 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 $s_{max}^i(\theta)$. $s_{max}^i(\Theta_h^i) = max\{s_{max}^i(\theta) : \forall\theta \in \Theta_h^i\}$.

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 job s_j^k is the "interfering job", and the job s_i^p is the "interfered job".

Due to *transitive retry* (introduced in Section III), 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 abort and 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. MOTIVATION

ECM [6], RCM [6], and LCM [7] suffer from *transitive retry*. Transitive retry is illustrated by the following example:

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, and s_2^y and s_3^z share objects. s_1^x and s_3^z do not share objects. Now, 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 will retry transitively because of s_3^z, which will increase the retry cost of s_1^x. Now, consider another atomic section s_4^f with a priority higher than that of s_3^z. Suppose s_4^f shares objects only with s_3^z. Thus, s_4^f can cause s_3^z to retry, which in turn will cause 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 higher priority tasks 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 $\Theta_z^h \cap \Theta_i^k = \emptyset$.

Therefore, the analysis in [6] and [7] extends the set of objects that can cause an atomic section of a lower priority job to retry. This is 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. The final set of objects (tasks) that can cause transactions in τ_i to retry is $\theta_i^{ex}(\gamma_i^{ex})$, respectively[1].

PNF [9], [10] is designed to avoid transitive retry by concurrently executing at most m non-conflicting transactions together. These executing transactions are non-preemptive. Thus, executing transactions cannot be aborted due to direct or indirect conflict with other transactions. However, with PNF, all objects accessed by each transaction must be known a-priori. Therefore, this is not suitable with dynamic STM implementations [11]. Additionally, PNF is implemented in [10] as a centralized CM that uses locks. This increases overhead.

Thus, we propose the *First Bounded, Last Timestamp contention manager* (or FBLT) that achieves the following goals:

1) reduce the retry cost of each transaction s_i^k due to another transaction s_j^l, just as LCM [7] does compared to ECM [6] and RCM [6].

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

2) avoid or bound the effect of transitive retry, similar to PNF [9], [10], without prior knowledge of accessed objects by each transaction, enabling dynamic STM.
3) decentralized design and avoid the use of locks, thereby reducing overhead.

IV. The FBLT Contention Manager

ALGORITHM 1: The FBLT Algorithm

Data: s_i^k: interfered transaction;
s_j^l: interfering transactions;
δ_i^k: the maximum number of times s_i^k can be aborted during T_i;
η_i^k: number of times s_i^k has already been aborted up to now;
m_set: contains at most m non-preemptive transactions. m is number of processors;
m_prio: priority of any transaction in m_set. m_prio is higher than any priority of any real-time task;
$r(s_i^k)$: time point at which s_i^k joined m_set;
Result: atomic sections that will abort

1 **if** s_i^k, $s_j^l \notin m\_set$ **then**
2   Apply LCM [7];
3   **if** s_i^k *is aborted* **then**
4   **if** $\eta_i^k < \delta_i^k$ **then**
5   Increment η_i^k by 1;
6   **else**
7   Add s_i^k to m_set;
8   Record $r(s_i^k)$;
9   Increase priority of s_i^k to m_prio;
10   **end**
11   **else**
12   Swap s_i^k and s_j^l;
13   Go to Step 3;
14   **end**
15 **else if** $s_j^l \in m\_set, s_i^k \notin m\_set$ **then**
16   Abort s_i^k;
17   **if** $\eta_i^k < \delta_i^k$ **then**
18   Increment η_i^k by 1;
19   **else**
20   Add s_i^k to m_set;
21   Record $r(s_i^k)$;
22   Increase priority of s_i^k to m_prio;
23   **end**
24 **else if** $s_i^k \in m\_set, s_j^l \notin m\_set$ **then**
25   Swap s_i^k and s_j^l;
26   Go to Step 15;
27 **else**
28   **if** $r(s_i^k) < r(s_j^l)$ **then**
29   Abort s_j^l;
30   **else**
31   Abort s_i^k;
32   **end**
33 **end**

Algorithm 1 illustrates FBLT. Each transaction s_i^k can be aborted during T_i for at most δ_i^k times. η_i^k records the number of times s_i^k has already been aborted up to now. If s_i^k and s_j^l have not joined the m_set yet, then they are preemptive transactions. Preemptive transactions resolve conflicts using LCM [7] (step 2). Thus, FBLT defaults to LCM when no transaction reaches its δ. If only one of the transactions is in the m_set, then the non-preemptive transaction (the one in m_set) aborts the other one (steps 15 to 26). η_i^k is incremented each time s_i^k is aborted as long as $\eta_i^k < \delta_i^k$ (steps 5 and 18). Otherwise, s_i^k is added to the m_ set and its priority is increased to m_prio (steps 7 to 9 and 20 to 22). When the priority of s_i^k is increased to m_prio, s_i^k becomes a non-preemptive transaction. Non-preemptive transactions cannot be aborted by other preemptive transactions, nor by any other real-time job. The m_set can hold at most m concurrent transactions because there are m processors in the system. $r(s_i^k)$ records the time s_i^k joined the m_set (steps 8 and 21). When non-preemptive transactions conflict together (step 27), the transaction with the smaller $r()$ commits first (steps 29 and 31). Thus, non-preemptive transactions are executed in FIFO order of the m_set.

V. Retry Cost and Response Time Bounds

We now derive an upper bound on the retry cost of any job τ_i^x under FBLT 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 1: The total retry cost for any job τ_i^x under FBLT due to 1) conflicts between its transactions and transactions of other jobs during an interval $L \leq T_i$ and 2) release of higher priority jobs is upper bounded by:

$$RC_{to}(L) \leq \sum_{\forall s_i^k \in s_i} \left(\delta_i^k len(s_i^k) + \sum_{\forall s_{iz}^k \in \chi_i^k} len(s_{iz}^k) \right) + RC_{re}(L) \quad (1)$$

where χ_i^k is the set of at most $m-1$ maximum length transactions conflicting directly or indirectly (through transitive retry) with s_i^k. Each transaction $s_{iz}^k \in \chi_i^k$ belongs to a distinct task τ_j. $RC_{re}(L)$ is the retry cost resulting from the release of higher priority jobs which preempt τ_i^x. $RC_{re}(L)$ is calculated by (6.8) in [10] for G-EDF, and (6.10) in [10] for G-RMA schedulers.

Proof: By the definition of FBLT, $s_i^k \in \tau_i^x$ can be aborted a maximum of δ_i^k times before s_i^k joins the m_set. Before joining the m_set, s_i^k can be aborted due to higher priority transactions, or transactions in the m_set. The original priority of transactions in the m_set can be higher or lower than p_i^x. Thus, the maximum time s_i^k is aborted before joining the m_set occurs if s_i^k is aborted for δ_i^k times.

Transactions preceding s_i^k in the m_set can conflict directly with s_i^k, or indirectly through transitive retry. The worst case scenario for s_i^k after joining the m_set occurs if s_i^k is preceded by $m-1$ maximum length conflicting transactions. Hence, in the worst case, s_i^k has to wait for the previous $m-1$ transactions to commit first. The priority of s_i^k after joining the m_set is higher than any real-time job. Therefore, s_i^k is not aborted by any job. If s_i^k has not joined the m_ set yet, and a higher priority job τ_j^y is released while s_i^k is running, then s_i^k may be aborted if τ_j^y has conflicting transactions with s_i^k. τ_j^y causes only one abort in τ_i^x because τ_j^y preempts τ_i^x only once. If s_i^k has already joined the m_set, then s_i^k cannot be aborted by the release of higher priority jobs. Thus, the maximum number of times transactions in τ_i^x can be aborted due to the release of higher priority jobs is less than or equal to the number of interfering higher priority jobs to τ_i^x. Claim follows. ∎

Claim 2: Under FBLT, the blocking time of a job τ_i^x due to lower priority jobs is upper bounded by:

$$D(\tau_i^x) = \sum \left(max_m(s_{j_{max}, \forall \tau_j^l,\, p_j^l < p_i^x}) \right) \quad (2)$$

where $s_{j_{max}}$ is the maximum length transaction in any job τ_j^l with original priority lower than p_i^x. The right hand side of (2) is the sum of the m maximum transactional lengths in all jobs with lower priority than τ_i^x.

Proof: The worst case blocking time for τ_i^x occurs when the maximum length m transactions in lower priority jobs than τ_i^x are executing non-preemptively. After commit of each transaction in the m_set, a higher priority job τ_j^y than τ_i^x is released. So, τ_j^y allocates the released processor instead of τ_i^x. Consequently, τ_i^x has to wait for the whole maximum length m transactions of lower priority jobs. Claim follows. ■

Claim 3: The response time of any job τ_i^x during an interval $L \le T_i$ under FBLT is upper bounded by:

$$R_i^{up} = c_i + RC_{to}(L) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} W_{ij}(R_i^{up}) \right\rfloor \quad (3)$$

where $RC_{to}(L)$ is calculated by (1), $D(\tau_i^x)$ is calculated by (2), and $W_{ij}(R_i^{up})$ is calculated by (11) in [6] for G-EDF, and (17) in [6] for G-RMA schedulers. (11) and (17) in [6] inflates c_j of any job $\tau_j^y \neq \tau_i^x$, $p_j^y > p_i^x$ by the retry cost of transactions in τ_j^y.

Proof: The response time of a job is calculated directly from FBLT's behavior. The response time of any job τ_i^x is the sum of its worst case execution time c_i, plus the retry cost of transactions in τ_i^x ($RC_{to}(L)$), plus the blocking time of τ_i^x ($D(\tau_i^x)$), and the workload interference of higher priority jobs. The workload interference of higher priority jobs scheduled by G-EDF is calculated by (11) in [6], and by (17) in [6] for G-RMA. Claim follows. ■

VI. Schedulability Comparison

We now (formally) compare the schedulability of FBLT against PNF [9], [10]. Toward this, we compare the total utilization under FBLT with that under PNF. In this comparison, we use the inflated execution time of the task, which is the sum of the worst-case execution time of the task and its retry cost, in the utilization calculation of the task.

Let $RC_A(T_i)$ and $RC_B(T_i)$ denote the retry cost of a job τ_i^x during T_i using the synchronization methods A and B, respectively. Let $D_A(\tau_i)$ and $D_B(\tau_i)$ be the maximum blocking time of any job τ_i^x due to lower priority jobs by methods A and B respectively. Now, schedulability of A is comparable to B if:

$$\sum_{\forall \tau_i} \frac{c_i + RC_A(T_i) + D_A(\tau_i)}{T_i} \le \sum_{\forall \tau_i} \frac{c_i + RC_B(T_i) + D_B(\tau_i)}{T_i}$$

$$\sum_{\forall \tau_i} \frac{RC_A(T_i) + D_A(\tau_i)}{T_i} \le \sum_{\forall \tau_i} \frac{RC_B(T_i) + D_B(\tau_i)}{T_i} \quad (4)$$

Claim 4: Schedulability of FBLT is equal or better than PNF if: 1) For each transaction s_i^k, maximum abort times δ_i^k equals at most the ratio between difference of total length of all transactions that can conflict only with s_i^k and total length of at most $m-1$ longest transactions that can conflict directly or transitively with s_i^k to length of s_i^k. 2) For any job τ_i^x, ratio between longest transaction in τ_i^x or lower priority jobs to smallest transaction in lower priority jobs equals at most the ratio between minimum number of times τ_i^x can be blocked due to non-conflicting transactions in all lower priority jobs to maximum release time of all jobs not belonging to τ_i.

Proof:

Substitute $RC_A(T_i)$ and $RC_B(T_i)$ in (4) with (1) and (3) in [9] respectively. Substitute $D_A(\tau_i)$ and $D_B(\tau_i)$ by (2) and (4) in [9] respectively. Substituting $RC_{re}(T_i) = \sum_{\forall \tau_j \in \zeta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) s_{i_{max}}$, covers $RC_{re}(T_i)$ given by (6.8) and (6.10) in [10] and maintains correctness of (4). $zeta_i$ is the set of higher priority tasks than any job of τ_i.

Let $\beta_i^1 = \sum_{\forall s_i^k \in s_i} \left(\delta_i^k len(s_i^k) + \sum_{s_{iz}^k \in \chi_i^k} len(s_{iz}^k) \right)$, $\beta_i^2 = \sum_{\forall \tau_j \in \zeta_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) s_{i_{max}} + \sum_{max\_m} \left\{ s_{j_{max}, \forall \bar{\tau_j^l}} \right\}$, $\beta_i^3 = \sum_{\forall \tau_j \in \gamma_i} \sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^h}(\theta)} len\left(\bar{s_j^h}(\theta) \right) \right)$ and $\beta_i^4 = \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h} \right) \right) \right\rfloor$. So, (4) holds if

$$\sum_{\forall \tau_i} \frac{\beta_i^1 + \beta_i^2}{T_i} \le \sum_{\forall \tau_i} \frac{\beta_i^3 + \beta_i^4}{T_i} \quad (5)$$

(5) holds if $\forall \tau_i$

$$\beta_i^1 + \beta_i^2 \le \beta_i^3 + \beta_i^4 \quad (6)$$

or $\forall \tau_i$

$$\beta_i^1 \le \beta_i^3 \quad and \quad \beta_i^2 \le \beta_i^4 \quad (7)$$

According to first part of (7)

$$\sum_{\forall s_i^k \in s_i} \left(\delta_i^k len(s_i^k) + \sum_{s_{iz}^k \in \chi_i^k} len(s_{iz}^k) \right) \quad (8)$$
$$\le \sum_{\forall \tau_j \in \gamma_i} \sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^h}(\theta)} len\left(\bar{s_j^h}(\theta) \right) \right)$$

For each $s_i^k \in s_i$, there are a set of zero or more $\bar{s_j^h}(\Theta) \in \tau_j, \forall \tau_j \neq \tau_i$ that are conflicting with s_i^k. Assuming this set of conflicting transactions with s_i^k is denoted as $\eta_i^k(j) = \left\{ \bar{s_j^h}(\Theta) \in \tau_j : (\Theta \in \theta_i) \wedge (\tau_j \neq \tau_i) \wedge \left(\bar{s_j^h}(\Theta) \notin \eta_i^l,\, l \neq k \right) \right\}$. The last condition $\bar{s_j^h}(\Theta) \notin \eta_i^l$, $l \neq k$ in definition of η_i^k ensures that common transactions $\bar{s_j^h}$ that can conflict with more than one transaction $s_i^k \in \tau_i$ are split among different $\eta_i^k(j)$, $k = 1, .., |s_i|$. This condition is necessary because in PNF, no two or more transactions of τ_i^x can be aborted by the same transaction of τ_j^h. Let γ_i^k be subset of γ_i that contains tasks with transactions conflicting directly with s_i^k. By substitution of $\eta_i^k(j)$ and γ_i^k in (8), (8) holds if $\forall s_i^k$:

$$\therefore \delta_i^k \le \frac{\sum_{\forall \tau_j \in \gamma_i^k} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^h}(\Theta) \in \eta_i^k(j)} len\left(\bar{s_j^h}(\theta) \right) \right)}{len(s_i^k)} - \frac{\sum_{s_{iz}^k \in \chi_i^k} len(s_{iz}^k)}{len(s_i^k)} \quad (9)$$

By definition of $\eta_i^k(j)$, if $\bar{s_j^h}(\Theta)$ can conflict with s_i^k and s_i^l, then $\bar{s_j^h}(\Theta)$ belongs either to η_i^k or η_i^l, but not both. Let

$\bar{\eta_i^k}(j) = \eta_i^k(j) - \left\{\bar{s_j^h}(\Theta) | \bar{s_j^h}(\Theta)\, can\, belong\, to\, \eta_i^l,\ l \neq k\right\}$. So, $\bar{\eta_i^k}(j)$ equals $\eta_i^k(j)$ excluding any transaction that can belong to another $\eta_i^l(j)$, $l \neq k$. (9) holds if

$$\delta_i^k \le \frac{\sum_{\forall\tau_j\in\gamma_i^k}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall \bar{s_j^h}(\Theta)\in\bar{\eta_i^k}(j)} len\left(\bar{s_j^h}(\theta)\right)\right)}{len(s_i^k)} - \frac{\sum_{s_{iz}^k\in\chi_i^k} len(s_{iz}^k)}{len(s_i^k)} \qquad (10)$$

Now, we consider the second part of (7). Let $s_{i,j_{max}} = max_{\forall\bar{\tau_j^l}}(s_{i_{max}}, s_{j_{max}})$. So, $s_{i,j_{max}}$ is the maximum transactional length in any job of τ_i or any lower priority job. Let $s_{j_{min}} = min\left\{len(s_j^h), \forall s_j^h \in \bar{\tau_j^l}\right\}$. So, $s_{j_{min}}$ is the smallest transactional length in any job of τ_j with lower priority than any job of τ_i. $\because$ $\sum_{max\_m}\left\{s_{j_{max},\forall\bar{\tau_j^l}}\right\} \le \sum_{\forall\bar{\tau_j^l}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)s_{i,j_{max}}$, $\therefore$ the second part of (7) holds if

$$\sum_{\forall\tau_j\in\zeta_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)s_{i,j_{max}} + \sum_{\forall\bar{\tau_j^l}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)s_{i,j_{max}} \le \left\lfloor\frac{1}{m}\sum_{\forall\bar{\tau_j^l}}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)|\ddot{s_j^h}|\,s_{j_{min}}\right)\right\rfloor \qquad (11)$$

$\because$ ζ_i includes all jobs with higher priority than τ_i, and $\bar{\tau_j^l}$ includes all jobs with lower priority than τ_i, $\therefore$ (11) holds if $\forall\,\tau_i$

$$\therefore \frac{s_{i,j_{max}}}{s_{j_{min}}} \le \frac{\left\lfloor\frac{1}{m}\sum_{\forall\bar{\tau_j^l}}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)|\ddot{s_j^h}|\right)\right\rfloor}{\sum_{\forall\tau_j\neq\tau_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)} \qquad (12)$$

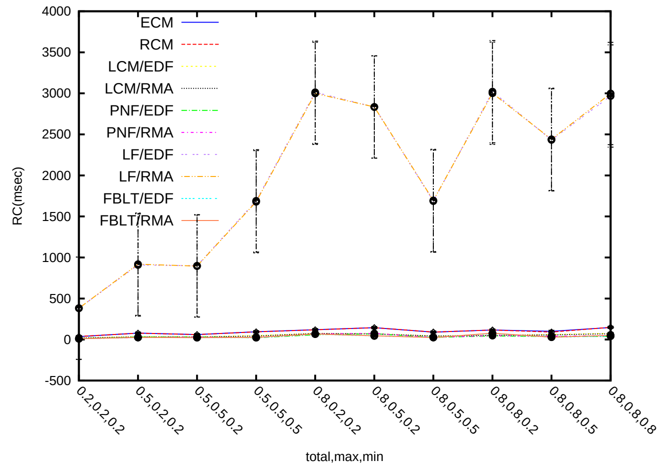
From (10) and (12), Claim follows.

■

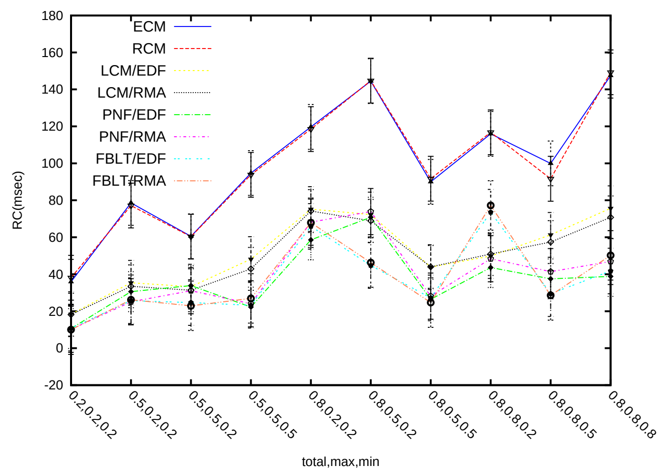
VII. EXPERIMENTAL EVALUATION

We now would like to understand how FBLT's retry cost compares with competitors in practice (i.e., on average). Since this can only be understood experimentally, we implement FBLT and the competitors and conduct experiments.

We used the ChronOS real-time Linux kernel [13] and the Rochester STM (RSTM) library [12] in our implementation. We implemented G-EDF and G-RMA schedulers in ChronOS, and modified RSTM to include implementations of FBLT, 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 three parameters: the maximum and minimum lengths of any atomic section within a task, and 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 FBLT's retry cost with that of lock-free (and other CMs) for one object per transaction. We then compare FBLT's retry cost with that of other CMs for multiple objects per transaction.



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



(b) ECM, RCM, LCM, PNF, FBLT

Fig. 1. Average retry cost (one object/transaction).

Figure 1 shows the average retry cost for the 5 task set sharing one object. On the x-axis of the figures, 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. The confidence level of all data points is 0.95. While Figure 1(a) includes all synchronization methods, Figure 1(b) excludes lock-free. From these figures, we observe that lock-free has the largest retry cost, as it provides no conflict resolution. FBLT has the largest retry cost among CMs, because transactions share only one object in this case. For multiple objects per transaction, PNF has an advantage over FBLT. However, PNF requires a-priori knowledge of all objects accessed by each transaction, whereas FBLT does not. Consequently, retry cost under PNF is a little shorter than that under FBLT. Experiments show that FBLT's retry cost can be shorter than that under ECM, RCM, and LCM, and can be comparable to that of PNF's as shown in Figure 2. PNF was designed to avoid transitive retry. Previous experiments compares retry cost of different CMs

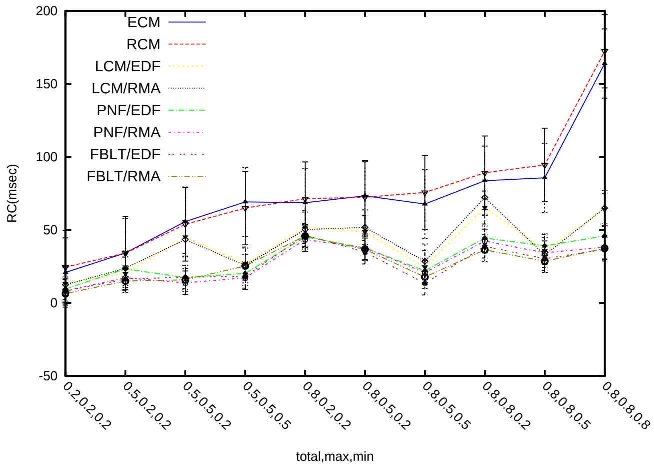


Fig. 2. Average retry cost (40 shared objects, 20 tasks).

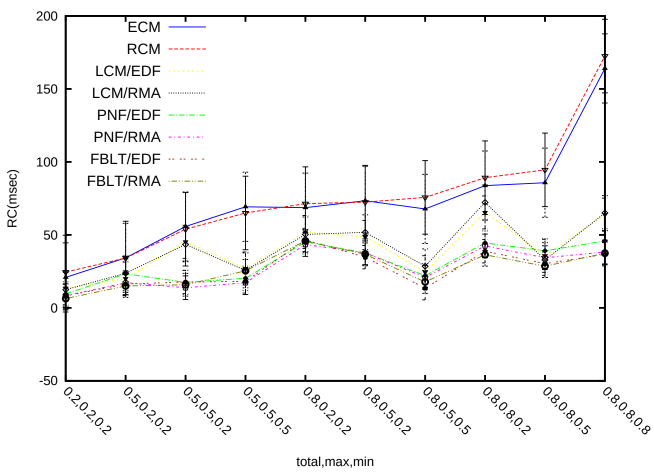


Fig. 3. Average retry cost (40 shared objects, 20 tasks).

in case of transitive retry. Figure 3 compares retry costs of different CMs in case of non-transitive retry. FBLT achieves shorter or comparable retry cost to other CMs including PNF. Similar trends were observed for the other task sets; those are omitted here due to space limitations.

VIII. Conclusions

Transitive retry increases transactional retry costs under ECM, RCM, and LCM. PNF avoids transitive retry by avoiding transactional preemptions. It avoids transitive retry cost by concurrently executing non-conflicting transactions, which are non-preemptive. However, PNF requires a-priori knowledge about objects accessed by each transaction. This is incompatible with dynamic STM implementations. Thus, we introduce the FBLT contention manager. Under FBLT, each transaction is allowed to abort for a no larger than a specified number of times. Afterwards, the transaction becomes non-preemptive. Non-preemptive transactions have higher priorities than other preemptive transactions and real-time jobs. Non-preemptive transactions resolve their conflicts using FIFO order. By proper adjustment of the maximum abort number of each transaction, we showed that FBLT's schedulability is equal to or better than PNF.

Our experimental results show that FBLT has equal or shorter retry cost than ECM, RCM, and LCM. PNF requires a-priori knowledge of all objects accessed by each transaction. This is an advantage for PNF over FBLT. Consequently, retry cost under PNF is shorter than that under FBLT in case of transitive retry. Still, FBLT's retry cost can be comparable to PNF's. In case of no or low transitive retry, FBLT achieves shorter retry cost than other CMs including PNF. Future work includes choosing another criterion to resolve conflicts of non-preemptive transactions. Also, using feedback from the system to adjust maximum abort number of each transaction. Consequently, retry cost can be reduced over time.

References

[1] M. Herlihy, "The art of multiprocessor programming," in *PODC*, 2006, pp. 1–2.

[2] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *PODC*, 2005, pp. 258–264.

[3] B. Saha, A.-R. Adl-Tabatabai *et al.*, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *PPoPP*, 2006, pp. 187–197.

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

[5] S. Fahmy and B. Ravindran, "On STM concurrency control for multicore embedded real-time software," in *International Conference on Embedded Computer Systems*, July 2011, pp. 1 –8.

[6] M. El-Shambakey and B. Ravindran, "STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs," in *Proceedings of the 27th SAC*. ACM, 2012, pp. 1602–1609.

[7] ——, "STM concurrency control for embedded real-time software with tighter time bounds," in *Proceedings of the 49th DAC*. ACM, 2012, pp. 437–446.

[8] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[9] M. Elshambakey and B. Ravindran, "On real-time STM concurrency control for embedded software with improved schedulability," *18th ASP-DAC*, 2013, (to appear).

[10] M. El-Shambakey, "Real-time software transactional memory: Contention managers, time bounds, and implementations," PhD Proposal, Virginia Tech, 2012, available as http://www.real-time.ece.vt.edu/shambakey_prelim.pdf.

[11] M. Herlihy *et al.*, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 22nd PODC*. ACM, 2003, pp. 92–101.

[12] Marathe *et al.*, "Lowering the overhead of nonblocking software transactional memory," in *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[13] M. Dellinger, P. Garyali, and B. Ravindran, "ChronOS Linux: a best-effort real-time multiprocessor linux kernel," in *Proceedings of the 48th DAC*. ACM, 2011, pp. 474–479.