STM Concurrency Control with Checkpointing for Embedded Real-Time Software with Tighter Time Bounds

Abstract

We consider checkpointing with software transactional memory (STM) concurrency control for embedded multicore real-time software, and present a modified version of FBLT contention manager called *Checkpointing FBLT* (CPFBLT). We upper bound transactional retries and task response times under CPFBLT, and identify when CPFBLT is a more appropriate alternative to FBLT without checkpointing.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-based Systems*]: Real-time and embedded systems

General Terms Design, Experimentation, Measurement

Keywords Software transactional memory (STM), real-time contention manager

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 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 [16]. 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) [14] 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 [25]), and is composable [15]. 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 proposed resolving transactional contention using dynamic and fixed priorities of parent threads. [7, 8, 11] present Earliest Deadline CM (ECM) and Rate Monotonic CM (RCM) , which are used with global EDF (G-EDF) and global RMS (G-RMS) multicore real-time schedulers [4]. In particular, [8] 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 and cannot handle multiple objects per transaction efficiently. These limitations are overcome with the Priority with Negative value and First access CM (PNF) [6, 10]. However, PNF requires prior knowledge of all objects accessed by each transaction. This significantly limits programmability, and is incompatible with dynamic STM implementations [17]. Additionally, PNF is a centralized CM, which increases overheads and retry costs, and has a complex implementation. First Bounded, Last Timestamp CM (or FBLT) [9], in contrast to PNF, does not require prior 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.

Checkpointing [20] can be used to further reduce response time of threads with conflicting transactions. Under checkpointing, a transaction retreats to a previous control flow location upon conflict. So, an aborted transaction does not have to retreat to its beginning.

We introduce checkpointing FBLT (CPFBLT) that extends original FBLT with checkpointing. (Section 5). We present the motivation for introducing checkpointing into FBLT (Section 4). We establish CPFBLT's retry and response time upper bounds under G-EDF and G-RMA schedulers (Section 6). We also identify the conditions under which CPFBLT is a better alternative to non-checkpointing FBLT (Section 7).

We implement FBLT and CPFBLT in the Rochester STM framework [22] and conduct experimental studies (Section 8). Our results reveal that CPFBLT has shorter response time than non-checkpointing FBLT.

Thus, the paper's contribution is the use of checkpointing as a complementary tool to FBLT to further enhance response time. CPFBLT contention manager with superior timeliness properties. CPFBLT, 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.

2. Related Work

Transactional-like concurrency control without using locks, for real-time systems, has been previously studied in the context of non-blocking data structures (e.g., [1]). Despite their numerous advantages over locks (e.g., deadlock-freedom), their programmability has remained a challenge. Past studies show that they are best suited for simple data structures where their retry cost is competitive to the cost of lock-based synchronization [3]. In contrast, STM is semantically simpler [16], and is often the only viable lock-free solution for complex data structures (e.g., red/black tree) [13] and nested critical sections [25]. STM concurrency control for real-time systems has been previously studied in [2, 7–10, 12, 13, 21, 26, 27].

[21] proposes a restricted version of STM for uniprocessors. [12] bounds response times in distributed systems with STM synchronization. [12] considers Pfair scheduling, limit to small atomic regions with fixed size, and limit transaction execution to span at most two quanta. [26] presents real-time scheduling of transactions and serializes transactions based on deadlines. However, the work does not bound retries and response times. [27] proposes real-time HTM. [27] assumes that the worst case conflict between atomic sections of different tasks occurs when the sections are released at the same time.

[13] upper bounds retries and response times for ECM with G-EDF, and identify the tradeoffs with locking and lock-free protocols. Similar to [27], [13] also assumes that the worst case conflict between atomic sections occurs when the sections are released simultaneously. The ideas in [13] are extended in [2], which presents three real-time CM designs.

[8] presents the ECM and RCM contention managers, and upper bounds transactional retries and task response times under them. [8] also identifies the conditions under which ECM and RCM are superior to lock-free techniques. In particular, [8] shows that, STM's superiority holds only under some ranges for the maximum atomic section length. Moreover, [8] restricts transactions to access only one object.

[7] presents the LCM contention manager, and upper bounds transactional retry cost and task response times for G-EDF and G-RMA schedulers. This work also compares (analytically and experimentally) LCM with ECM, RCM, and lock-free synchronization. However, similar to [7], [8] restricts transactions to access only one object.

[10] presents the PNF contention manager, which allows transactions to access multiple objects and avoids the consequent transitive retry effect. The work also upper bounds transactional retries and task response times under G-EDF and G-RMA. However, PNF requires a-priori knowledge of the objects accessed by each transaction, which is not always possible, limits programmability, and is incompatible with dynamic STM implementations [17]. Additionally, PNF is a centralized CM and uses locks in its implementation, which increases overheads.

[9] presents the FBLT contention manager. In contrast to PNF, FBLT does not require prior knowledge of required objects by each transaction. FBLT premits multiple objects per transaction. Under FBLT, each transaction can be aborted for a specific number of times. Afterwards, the transaction becomes non-preemptive. Non-preemptive transaction cannot be aborted except by another non-preemptive transaction. Non-preemptive transactions resolve conflicts based on the time they became non-preemptive.

Previous CMs try to enhance response time of real-time tasks using different policies for conflict resolution. Checkpointing does not require aborted transaction to restart from beginning. Thus, Checkpointing can be plugged into different CMs to further improve response time. [20] introduces checkpointing as an alternative to closed nesting transactions[29]. [20] uses boosted transactions [18] instead of closed nesting [19, 23, 29] to implement checkpointing. Booseted transactions are based on linearizable objects with abstract states and concrete implementation. Methods under boosted transaction have well defined semantics to transit objects from one state to another. Inverse methods are used to restore objects to previous states. Upon a conflict, a transaction does not need to revert to its beginning, but rather to a point where the conflict can be avoided. Thus, checkpointing enables partial abort. [28] applies check pointing in distributed transactional memory using Hyflow [24]. Checkpointing showed performance improvement compared to flat transactions.

3. Preliminary

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. 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 $\Theta = \{\theta \in \Theta_i^k\}$. $\bar{s}_i^k = \bar{s}_i^k(\Theta)$ 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 $\gamma_i(\theta)$.

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 transactions. 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* [9, 10], 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. $\Theta_i^{k^{ex}}$ is the subset of objects in Θ_i^{ex} that can cause direct or transitive conflict to s_i^k. 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. γ_i^k is the set of tasks that can directly cause s_i^k to abort and retry. $\gamma_i^{k^{ex}}$ is the set of tasks that can directly or indirectly (through transitive retry) cause s_i^k 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 $W_{ij}(L)$.

4. Motivation

Under checkpointing, a transaction $s_i^k \in \tau_i$ does not need to restart from the beginning upon a conflict on object θ. s_i^k just needs to return to the first point it accessed θ. Thus, response time of τ_i can be improved by checkpointing unless s_i^k acquires all its objects at its beginning. While the CM tries to resolve conflicts using proper strategies, checkpointing enhances performance by reducing aborted part of each transaction. Thus, checkpointing acts as a complementary component to different CMs to further enhance response time.

Behaviour of some CMs, like PNF [10], can make checkpointing useless. PNF requires a priori knowledge of accessed objects within transactions. Only the first m non-conflicting transactions are allowed to execute concurrently and non-preemptively. Thus, PNF makes no use of checkpointing because there is no conflict between non-preemptive transactions.

Other CMs (e. g., FBLT[9]) allow conflicting transaction to run concurrently. So, FBLT can benefit from checkpointing. FBLT, by definition, depends on LCM. LCM, in turn, depends on ECM (RCM) for G-EDF (G-RMA), respectively. Experimental results show superiority of FBLT over LCM, ECM and RCM[9]. Thus, we extend FBLT to checpointing FBLT (CPFBLT) to improve response time than the non-checkpointing FBLT (NCPFBLT).

5. Checkpointing FBLT (CPFBLT)

CPFBLT depends on FBLT which in turn depends on LCM [7]. So, we initially illustrate LCM with required modification to implement checkpointing (Section 5.1). Afterwards, we illustrate FBLT with checkpointing extension in (Section 5.2).

5.1 Checkpointing LCM (CPLCM)

CPLCM is shown in Algorithm 1. A new checkpoint is recorded for each newly accessed object θ by any transaction s_h^u (step 2). Checkpoint is recorded when θ is accessed for the first time because any further changes to θ will be discarded upon conflict. CPLCM uses the remaining length of s_i^k when it is interfered, as well as $len(s_j^l)$, to decide which transaction must be aborted. If $p_i^k > p_j^l$, then s_j^l would be the transaction to abort because of lower priority of s_j^l, and s_i^k started before s_j^l (step 5). Otherwise, c_{ij}^{kl} is calculated (step 8) to determine whether it is worth aborting s_i^k in favour of s_j^l. If $len(s_j^l)$ is relatively small compared to $len(s_i^k)$, then c_{ij}^{kl}, and α_{ij}^{kl} tend to be small (steps 8, 9). Consequently, s_i^k aborts in favour of s_j^l. Also, if the remaining execution length of s_i^k is long, then α tends to be small (step 10). Consequently, s_i^k will abort in favour of s_j^l. When s_i^k aborts upon a conflict with s_j^l on object θ_{ij}^{kl}, then checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$ are removed (step 13).

Algorithm 1: CPLCM

Data:
$s_i^k \rightarrow$ interfered transaction.
$s_j^l \rightarrow$ interfering transaction with s_i^k on object θ_{ij}^{kl}.
$\psi \rightarrow$ predefined threshold $\in [0,1]$.
$\varepsilon_i^k \rightarrow$ remaining execution length of $\{s_i^k\}$.
$cp_h^u(\theta) \rightarrow$ recorded checkpoint in transaction s_h^u for newly accessed object θ
Result: which transaction of s_i^k or s_j^l aborts

1 **foreach** *newly accessed* θ *requested by any transaction* s_h^u **do**
2   Add a checkpoint $cp_h^u(\theta)$
3 **end**
4 **if** $p_i^k > p_j^l$ **then**
5   s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
6   Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$
7 **else**
8     $c_{ij}^{kl} = len(s_j^l)/len(s_i^k)$;
9     $\alpha_{ij}^{kl} = ln(\psi)/(ln(\psi) - c_{ij}^{kl})$;
10     $\alpha = \left(len(s_i^k) - \varepsilon_i^k\right)/len(s_i^k)$;
11   **if** $\alpha \le \alpha_{ij}^{kl}$ **then**
12   s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
13   Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$
14   **else**
15   s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
16   Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$
17   **end**
18 **end**

Prior checkpoints to $cp_i^k(\theta_{ij}^{kl})$ remain the same. Also, if s_j^l aborts in favour of s_i^k, then all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$ are removed (steps 6, 16).

5.2 Design of CPFBLT

Algorithm 2 illustrates CPFBLT. A new checkpoint is recorded for each newly accessed object θ by any transaction s_h^u (step 2). Checkpoint is recorded when θ is accessed for the first time because any further changes to θ will be discarded upon conflict. 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 CPLCM (step 5). Thus, CPFBLT defaults to CPLCM when the conflicting transactions (s_i^k and s_j^l) have not reached their δs (δ_i^k and δ_j^l). η_i^k is incremented each time s_i^k is aborted as long as $\eta_i^k < \delta_i^k$ (steps 8 and 22). Otherwise, s_i^k is added to the m_ set and priority of s_i^k is increased to m_prio (steps 10 to 12 and 24 to 26). 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 (steps 18 to 30). 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 11 and 25). When non-preemptive transactions conflict together (step 31), the transaction that joined m_set first becomes the transaction that commits first (steps 33 and 36). Thus, non-preemptive transactions are executed in FIFO order. When $s_i^k(s_j^l)$ aborts due to a conflict on θ_{ij}^{kl} with $s_j^l(s_i^k)$, then $s_i^k(s_j^l)$ retreats to $cp_i^k(\theta_{ij}^{kl})(cp_j^l(\theta_{ij}^{kl}))$, respectively. All checkpoints recorded after $cp_i^k(\theta_{ij}^{kl})(cp_j^l(\theta_{ij}^{kl}))$ are removed (steps 20, 34 and 37).

Algorithm 2: The CPFBLT Algorithm

Data:
s_i^k: interfered transaction.
s_j^l: interfering transaction.
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.
$cp_h^u(\theta) \rightarrow$ recorded checkpoint in transaction s_h^u for newly accessed object θ
Result: which transaction, s_i^k or s_j^l, aborts

1 **foreach** *newly accessed* θ *requested by any transaction* s_h^u **do**
2   Add a checkpoint $cp_h^u(\theta)$
3 **end**
4 **if** $s_i^k, s_j^l \notin m\_set$ **then**
5   Apply CPLCM (Algorithm 1);
6   **if** s_i^k *is aborted* **then**
7   **if** $\eta_i^k < \delta_i^k$ **then**
8   Increment η_i^k by 1;
9   **else**
10   Add s_i^k to m_set;
11   Record $r(s_i^k)$;
12   Increase priority of s_i^k to *m_prio*;
13   **end**
14   **else**
15   Swap s_i^k and s_j^l;
16   Go to Step 6;
17   **end**
18 **else if** $s_j^l \in m\_set, s_i^k \notin m\_set$ **then**
19   s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
20   Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$;
21   **if** $\eta_i^k < \delta_i^k$ **then**
22   Increment η_i^k by 1;
23   **else**
24   Add s_i^k to m_set;
25   Record $r(s_i^k)$;
26   Increase priority of s_i^k to *m_prio*;
27   **end**
28 **else if** $s_i^k \in m\_set, s_j^l \notin m\_set$ **then**
29   Swap s_i^k and s_j^l;
30   Go to Step 18;
31 **else**
32   **if** $r(s_i^k) < r(s_j^l)$ **then**
33   s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
34   Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$;
35   **else**
36   s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
37   Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$;
38   **end**
39 **end**

6. CPFBLT Retry Cost

Claim 1. *Assume only two transaction s_i^k and s_j^l conflicting together. Let s_i^k begins at time $S\left(s_i^k\right)$ and s_j^l begins at time $S\left(s_j^l\right)$. Let $\triangle = S\left(s_j^l\right) - S\left(s_i^k\right)$. In the absence of checkpointing, retry cost of s_i^k due to s_j^l is given by*

$$BASE\_RC_{ij}^{kl} \leq \begin{cases} len\left(s_j^l\right) + \triangle & , -len\left(s_j^l\right) \leq \triangle \leq len\left(s_i^k\right) \\ 0 & , Otherwise \end{cases} \quad (1)$$

$BASE\_RC_{ij}^{kl}$ is upper bounded by

$$len\left(s_j^l\right) + \left(s_i^k\right) \quad (2)$$

which is the same upper bound given by Proofs of Claims 1 and 3 in [8]

Proof. Due to absence of checkpointing, s_i^k aborts and retries from its beginning due to s_j^l. So, s_i^k retries for the period starting from $S\left(s_i^k\right)$ to the end of execution of s_j^l. s_j^l ends execution at $S\left(s_j^l\right) + len\left(s_j^l\right)$. If $S\left(s_j^l\right) < S\left(s_i^k\right) - len\left(s_j^l\right)$, then s_j^l finishes execution before start of s_i^k and there will be no conflict. Also, if $S\left(s_j^l\right) > S\left(s_i^k\right) + len\left(s_i^k\right)$, then s_j^l starts execution after s_i^k finishes execution and there will be no conflict. Thus, (1) follows. Equation (2) is derived by substitution of $\triangle$ by its maximum value (i.e., $\left(s_i^k\right)$). Claim follows. □

Claim 2. *Assume only two transactions s_i^k and s_j^l conflicting on one object θ. Let ∇_j^l be the time interval between the start of s_j^l and the first access to θ. Similarly, let ∇_i^k be the time interval between the start of s_i^k and the first access to θ. Let $\triangle$ be the time difference between start of s_j^l relative to start of s_i^k. So, $\triangle < 0$ if s_j^l starts before s_i^k. Under checkpointing, s_i^k aborts and retries due to s_j^l for*

$$RC0_{ij}^{kl} \leq \begin{cases} len\left(s_j^l\right) - \nabla_i^k + \triangle & , if \begin{array}{l} \triangle \geq \nabla_i^k - len\left(s_j^l\right) \\ \triangle \leq len\left(s_i^k\right) - \nabla_j^l \end{array} \\ 0 & , Otherwise \end{cases} \quad (3)$$

$RC0_{ij}^{kl}$ is upper bounded by

$$len\left(s_j^l + s_i^k\right) - \nabla_j^l - \nabla_i^k \quad (4)$$

Proof. As s_i^k and s_j^l conflict only on one object θ, there will be no conflict before both s_i^k and s_j^l access θ. Retry cost of s_i^k due to s_j^l is derived by Claim 1 excluding parts of s_i^k and s_j^l before both transactions access θ. Thus, $len\left(s_i^k\right)$ in Claim 1 is substituted by $len\left(s_i^k\right) - \triangle_i^k$. $len\left(s_j^l\right)$ is substituted by $len\left(s_j^l\right) - \triangle_j^l$. $\triangle$ in Claim 1 is substituted by $\triangle + \nabla_j^l - \nabla_i^k$. Claim follows. □

Claim 3. *Assume only two transactions s_i^k and s_j^l conflicting on a number of object $\theta_1, \theta_2 \ldots \theta_z$. Let ∇_{i*}^k be the time interval between start of s_i^k and the first object θ_i accessed by s_i^k and shared with s_j^l. Let ∇_{j*}^l be the time interval between start of s_j^l and the first object θ_j accessed by s_j^l and shared with s_i^k. θ_i and θ_j may not be the same. With checkpointing, retry cost of s_i^k due to s_j^l is upper* *bounded by*

$$RC1_{ij}^{kl} \leq len\left(s_i^k + s_j^l\right) - \nabla_{i*}^k - \nabla_{j*}^l \tag{5}$$

Proof. Proof follows directly from Claim 2 by maximizing (4). $len\left(s_i^k\right)$, as well as, $len\left(s_j^l\right)$ in (4) cannot be changed. Thus, by choosing minimum values of ∇_i^k and ∇_j^l that correspond to shared objects between s_i^k and s_j^l, (4) is maximized. Claim follows. □

Claim 4. *If s_j^l is conflicting indirectly (through transitive retry) with s_i^k, then it is safe to ignore ∇_{i*}^k in calculating the upper bound of retry cost of s_i^k due to s_j^l.*

Proof. If s_j^l is conflicting indirectly with s_i^k, then s_j^l is accessing an object θ that does not belong to Φ_i^k. In this case, to get an upper bound for the retry cost of s_i^k due to s_j^l, ∇_{i*}^k assumes its minimum value in (5). Thus, $\nabla_{i*}^k = 0$. Claim follows. □

Claim 5. *Assume only two non-preemptive transactions s_i^k and s_j^l under CPFBLT. With checkpointing, retry cost of s_i^k due to direct or indirect conflict with s_j^l is upper bounded by*

$$RC2_{ij}^{kl} \leq len\left(s_j^l\right) - \nabla_{i*}^k \tag{6}$$

where $\nabla_{i*}^k = 0$ in case of indirect conflict.

Proof. Proof follows directly from Claims 2, 3 and 4 except that s_j^l must have become non-preemptive before s_i^k. So, s_j^l starts execution non-preemptively before s_i^k. Otherwise, by definition of CPFBLT, s_j^l will not be able to abort s_i^k. Thus, $\triangle$ must not exceed 0. Claim follows. □

Claim 6. *Let s_i^k be a non-preemptive transaction under CPFBLT. Let χ_i^k be the set of transactions conflicting (directly or indirectly) with s_i^k. Each transaction $s_j^l \in \chi_i^k$ belongs to a distinct task. Transactions in χ_i^k are organized in non-increasing order of $RC2_{ij}^{kl}$ for each $s_j^l \in \chi_i^k$. Total retry cost of non-preemptive transaction s_i^k due to other non-preemptive transactions is upper bounded by*

$$RC3_i^k \leq \sum_{a=1}^{a=min\left(|\chi_i^k|,m-1\right)} RC2_i^k\left(\chi_i^k(a)\right) \tag{7}$$

where $\chi_i^k(a)$ is the a^{th} transaction in χ_i^k. If $\chi_i^k(a) = s_j^l$, then $RC2_i^k\left(\chi_i^k(a)\right) = RC2_{ij}^{kl}$.

Proof. By definition of CPFBLT, a transaction s_i^k can be preceded by at most $m-1$ non-preemptive transactions. As non-preemptive transactions are organized in FIFO order, no two non-preemptive transactions can belong to the same task. Maximum retry cost of non-preemptive s_i^k occurs when: 1) s_i^k is preceded by at most $m-1$ transactions conflicting with s_i^k. 2) Each conflicting transaction s_j^l to s_i^k must have one of the highest $m-1$ values for $RC2_{ij}^{kl}$. 3) Non-preemptive transactions preceding s_i^k are executing sequentially. Thus, retry cost of non-preemptive s_i^k can be upper bounded by Claim 5 for at most the first $m-1$ transactions in χ_i^k. If the third condition is not satisfied, then (7) still gives a correct, but not tight, upper bound. Claim follows. □

Claim 7. *Under CPFBLT, a preemptive transaction s_i^k aborts and retries for at most*

$$RC4_i^k \leq \delta_i^k\left(len\left(s_i^k\right) - min\left(\nabla_{i*}^k\right)\right) \tag{8}$$

where $min\left(\nabla_{i*}^k\right)$ is the minimum ∇_{i*}^k for s_i^k and any other conflicting transaction s_j^l. If there are indirectly conflicting transactions with s_i^k, then $min\left(\nabla_{i*}^k\right) = 0$.

Proof. No transaction will make preemptive s_i^k aborts and retries before $min\left(\nabla_{i*}^k\right)$. By checkpointing, s_i^k will not retreat earlier than $min\left(\nabla_{i*}^k\right)$. By definition of CPFBLT, preemptive s_i^k aborts for at most δ_i^k times before it becomes non-preemptive. Claim follows. □

Claim 8. *The total retry cost of any job τ_i^x under CPFBLT due to 1) conflicts with other transactions during an interval L. 2) release of higher priority jobs during execution of preemptive transactions is upper bounded by*

$$RC(L)_{to}^i = \sum_{s_i^k \in s_i}\left(RC4_i^k + RC3_i^k\right) + RC_{re}(L) \tag{9}$$

where $RC_{re}(L)$ is the retry cost resulting from the release of higher priority jobs during execution of preemptive transactions. $RC_{re}(L)$ is calculated by (6.8) in [10] for G-EDF, and (6.10) in [10] for G-RMA schedulers.

Proof. Following Claims 4, 6, 7 and Claim 1 in [9], Claim follows. □

Any newly released task τ_i^x can be blocked by m lower priority non-preemptive nested transactions. Blocking time D_i of any job τ_i^x is independent of checkpointing. Thus, D_i is calculated by Claim 3 in [9]. Claim 2 in [9] is used to calculate response time under CPFBLT where $RC_{to}(T_i)$ is calculated by (9).

7. CPFBLT vs. NCPFBLT

Claim 9. *Schedulability of CPFBLT is better or equal to schedulability of NCPFBLT if shared objects within each transaction s_i^k are accessed lately relative to start of s_i^k.*

Proof. Let upper bound on retry cost of any task τ_i^x during T_i under NCPFBLT be denoted as RC_i^{ncp}. RC_i^{ncp} is calculated by Claim 1 in [9]. Let upper bound on retry cost of any task τ_i^x during T_i under CPFBTL be denoted as RC_i^{cp}. RC_i^{cp} is calculated by (9). Let D_i be the upper bound on blocking time of any newly released task τ_i^x during T_i due to lower priority jobs. D_i is the same for both CPFBLT and NCPFBLT. D_i is calculated by Claim 2 in [9]. For CPFBLT schedulability to be better than schedulability of NCPFBLT:

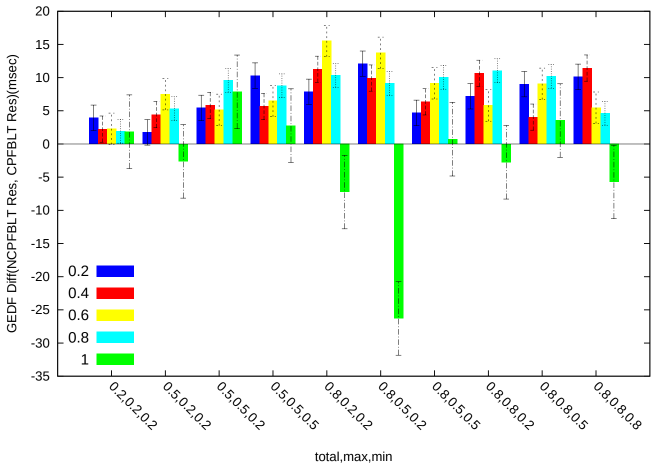
$$\sum_{\forall\tau_i}\frac{c_i + RC_i^{cp} + D_i}{T_i} \leq \sum_{\forall\tau_i}\frac{c_i + RC_i^{ncp} + D_i}{T_i} \tag{10}$$

$\because D_i$ and c_i are the same for each τ_i under CPFBLT and NCPFBLT, then (10) holds if:

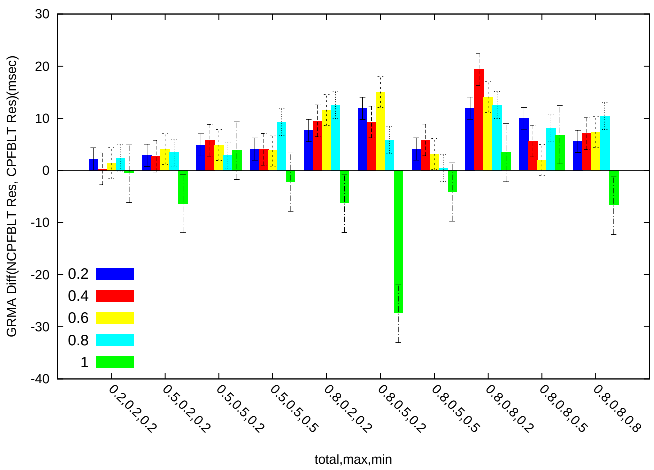
$$\forall\tau_i, RC_i^{cp} \leq RC_i^{ncp}$$

$$\delta_i^k\left(len\left(s_i^k\right) - min\left(\nabla_{i*}^k\right)\right) + \sum_{a=1}^{min\left(|\chi_i^k|,m-1\right)}\left(len\left(\chi_i^k(a)\right) - \nabla_{i*}^k\right)$$
$$\leq \delta_i^k len\left(s_i^k\right) + \sum_{a=1}^{min\left(|\gamma_i^k|,m-1\right)}\left(len\left(\gamma_i^k(a)\right)\right) \tag{11}$$

where γ_i^k is the set of at most $m-1$ longest transactions conflicting directly or indirectly with s_i^k. Thus, $\gamma_i^k(a) \geq \chi_i^k(a), \forall a$. Thus, by increasing ∇_{i*}^k, (11) holds. Claim follows. □

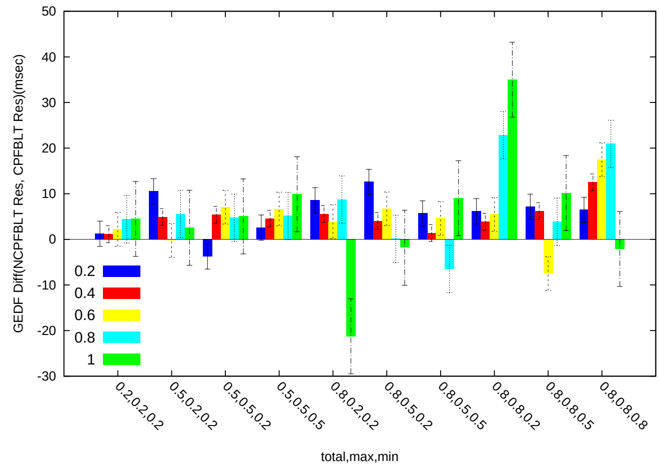


(a) 4 tasks, G-EDF, Response time

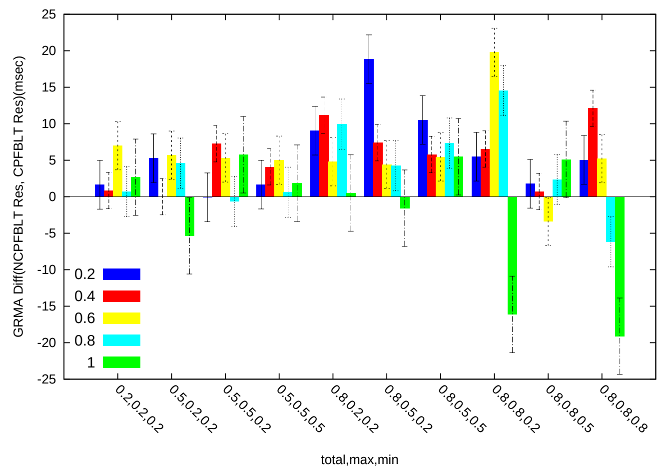


(b) 4 tasks, G-RMA, Response time

Figure 1. Average response time difference between NCPFBLT and CPFBLT for 4 tasks.



(a) 8 tasks, G-EDF, Response time



(b) 8 tasks, G-RMA, Response time

Figure 2. Average response time difference between NCPFBLT and CPFBLT for 8 tasks.

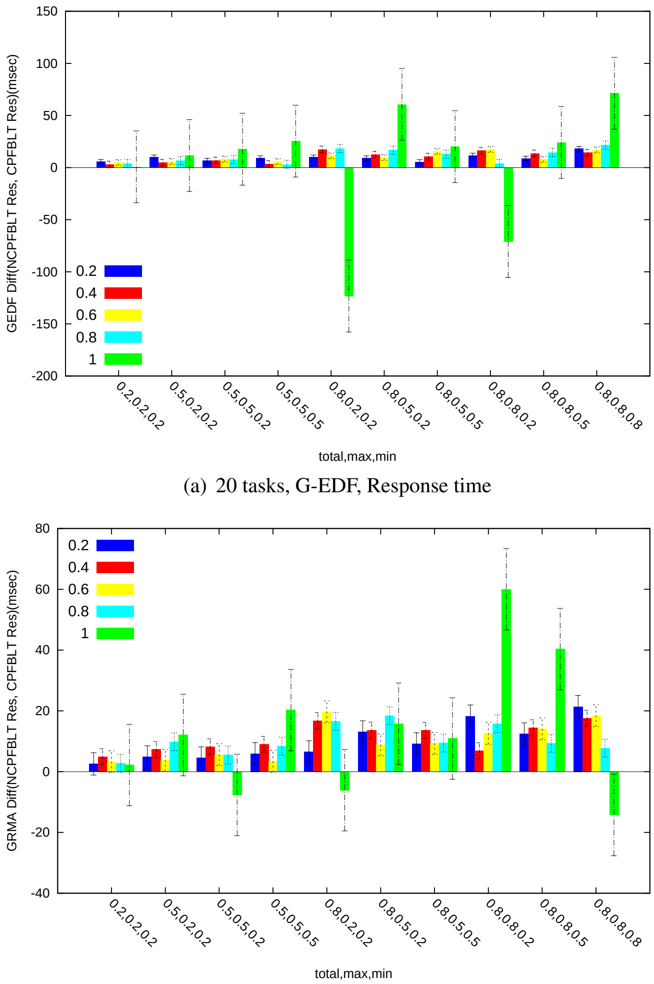
8. Experimental Evaluation

We now would like to understand how CPFBLT's retry cost and response time compare with NCPFBLT in practice (i.e., on average). Since this can only be understood experimentally, we implement CPFBLT and NCPFBLT and conduct experiments.

We used the ChronOS real-time Linux kernel [5] and the Rochester STM (RSTM) library [22] in our implementation. We implemented G-EDF and G-RMA schedulers in ChronOS, and modified RSTM to include implementations of CPFBLT and NCPFBLT. We used an 8 core, 2GHz AMD Opteron platform. We used three task sets consisting of 4, 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. For each run, $max(1-\nabla_i^k)_{\forall s_i^k} \in \{0, 0.2, 0.4, 0.6, 0.8\}$. $max(1-\nabla_i^k)$ represents the maximum transactional length ratio after which objects can be shared between transactions.

Average response time difference between NCPFBLT and CPFBLT for the 4, 8 and 12 task sets are shown in figures 1(a) to 3(b) for both G-EDF and G-RMA. Results show benefits of checkpointing when combined with FBLT. As level of sharing extends to all objects, this means that any transaction s_i^k under both CPFBLT and NCPFBLT can retreat to its beginning. Thus, in this case (i.e., objects can be shared from the beginning of each transaction), NCPFBLT can show some better results than CPFBLT as shown in the figures.

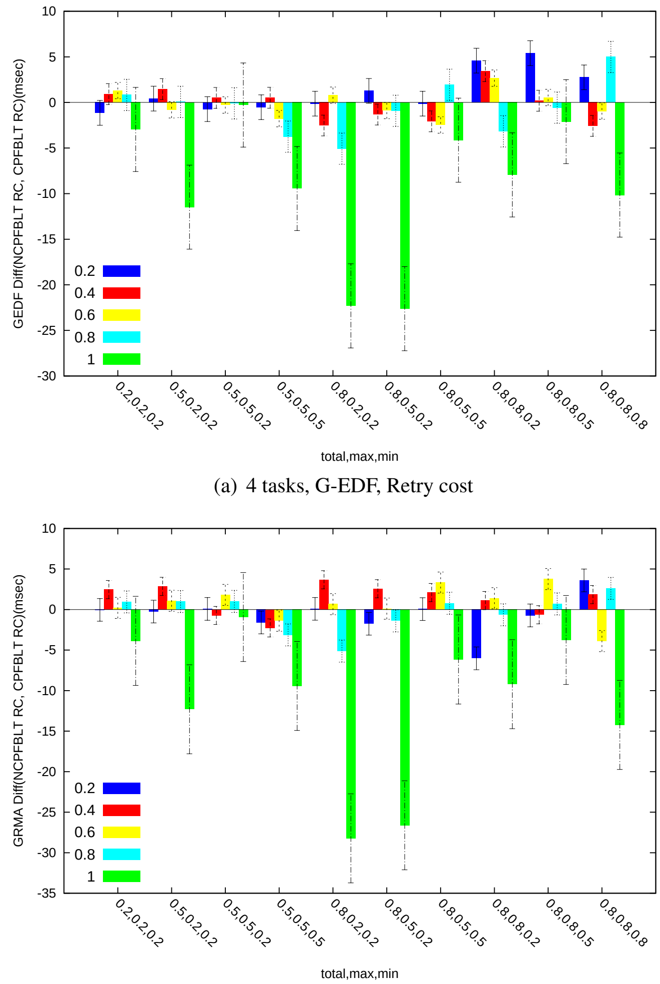
Figures 4(a) to 6(b) show average retry cost difference between NCPFBLT and CPFBLT. Retry cost results can be misleading because of the negative difference between retry cost of NCPFBLT and CPFBLT. But this is natural due to behaviour of CPFBLT. Under NCPFBLT, a transaction s_i^k returns to its beginning upon a conflict with s_j^l on object θ. Whereas, under CPFBLT, s_i^k returns to the first point it accessed θ. Thus, under CPFBLT, s_i^k tries to access θ directly after returning to the proper checkpoint. But s_j^l is still holding θ. Accordingly, s_i^k will abort and retry again. This retrial (donated as $RC_{cp}s_i^k$) is added to the accumulated retry cost of s_i^k under CPFBLT. Under NCPFBLT, s_i^k returns to its beginning when it conflicts with s_j^l. Thus, when s_i^k reaches θ once again, s_j^l may have finished execution. Let the time between start of s_i^k and first access to θ be $\nabla_i^k(\theta)$. Accordingly, retry cost of s_i^k under NCPFBLT



(a) 20 tasks, G-EDF, Response time

(b) 20 tasks, G-RMA, Response time

Figure 3. Average response time difference between NCPFBLT and CPFBLT for 20 tasks.



(a) 4 tasks, G-EDF, Retry cost

(b) 4 tasks, G-RMA, Retry cost

Figure 4. Average retry cost difference between NCPFBLT and CPFBLT for 4 tasks.

can be less than retry cost of s_i^k under CPFBLT. Whereas, $RC_{cp}s_i^k$ can be much less than $\nabla_i^k(\theta)$. Thus, $RC_{cp}s_i^k$ contributes by a smaller value to the response time of τ_i, in contrast to $\nabla_i^k(\theta)$. This why response time for CPFBLT is better than NCPFBLT, whereas, retry cost of NCPFBLT is less than CPFBLT.

9. Conclusion

Past research on real-time CMs focused on devloping different conflict resultion strategis for transactions. Except for LCM [7], no policy was made to reduce the length of conflicting transactions. In this paper, we analysed effect of checkpointing over FBLT CM. Analysis shows that response time of CPFBLT can be reduced than NCPFBLT by proper delaying access to shared objects. Experimental evaluation reveals better response time for CPFBLT than NCPFBLT. Despite retry cost of NCPFBLT is lower than retry cost of CPFBLT, but this natural as explained previously. Some CMs make no use of checkpointing due to behaviour of that CM (e.g, under PNF, all non-preemptive transactions are non-conflicting).

References

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

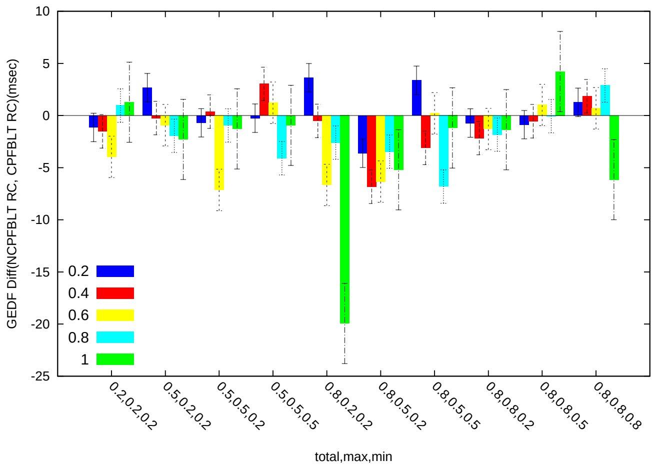
[2] A. Barros and L. Pinho. Managing contention of software transactional memory in real-time systems. In *IEEE RTSS, Work-In-Progress*, 2011.

[3] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 342 –353, april 2008.

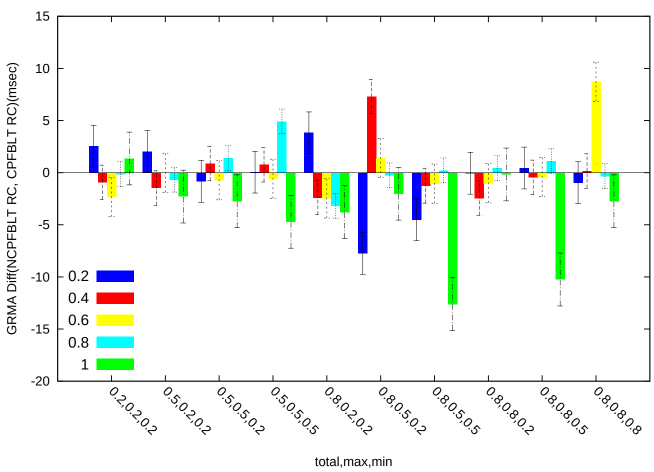
[4] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.

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

[6] 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`.

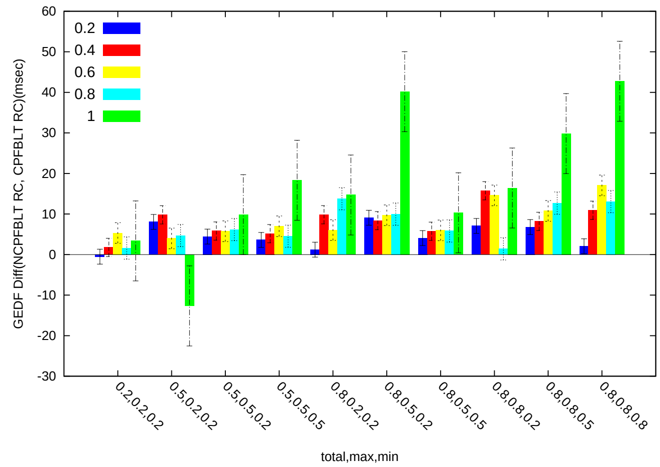


(a) 8 tasks, G-EDF, Retry cost

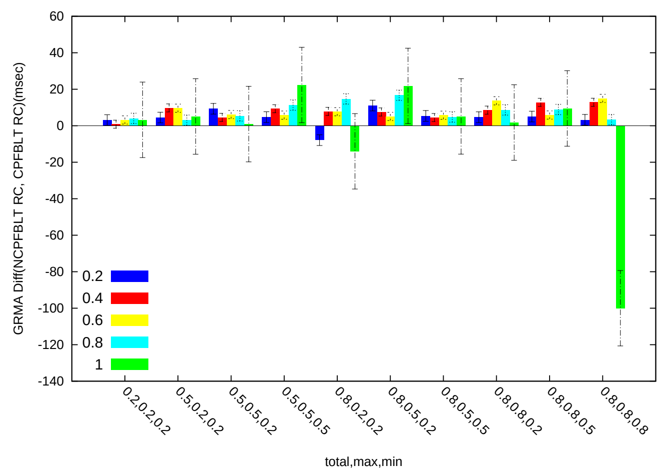


(b) 8 tasks, G-RMA, Retry cost

Figure 5. Average retry cost difference between NCPFBLT and CPFBLT for 8 tasks.



(a) 20 tasks, G-EDF, Retry cost



(b) 20 tasks, G-RMA, Retry cost

Figure 6. Average retry cost difference between NCPFBLT and CPFBLT for 20 tasks.

[7] M. El-Shambakey and B. Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *Proceedings of the 49th DAC*, pages 437–446. ACM, 2012.

[8] 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*, pages 1602–1609. ACM, 2012.

[9] M. Elshambakey and B. Ravindran. FBLT: A real-time contention manager with improved schedulability. *DATE*, 2013. (to appear).

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

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

[12] S. 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.

[13] S. F. Fahmy. *Collaborative Scheduling and Synchronization of Distributable Real-Time Threads*. PhD thesis, Virginia Tech, 2010.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

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

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

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

[18] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[19] J. Kim and B. Ravindran. Scheduling closed-nested transactions in distributed transactional memory. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 179 –188, may 2012.

[20] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 160–168, New York, NY, USA, 2008. ACM.

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

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

[23] S. Peri and K. Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *Proceedings of the 12th international conference on Distributed computing and networking*, pages 95–106. Springer-Verlag, 2011.

[24] M. M. Saad and B. Ravindran. Hyflow: a high performance distributed software transactional memory framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

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

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

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

[28] A. Turcu. *On Improving Distributed Transactional Memory Through Nesting and Data Partitioning*. Phd proposal, Virginia Tech, 2012. Available as `http://www.ssrg.ece.vt.edu/theses/PhdProposal_Turcu.pdf`.

[29] A. Turcu, B. Ravindran, and M. Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.