Investigating Real-Time Specifications for C#

Towards Integrating Hard Real-Time Capabilities with Time-Oblivious Applications in C#

Flaviu ROMAN
LPD, I&C, EPFL

***Abstract*—Nowadays, an increasing number of problems would benefit from solutions integrating the capabilities of both hard real-time and of normal, time-oblivious applications, running under the same environment. There is a need for abstractions and types, as well as a supporting virtual machine and memory management, which can provide guarantees for hard real-time tasks, while keeping the mainstream application running under the same Operating System transparent to the hard real-time features. In C# / .net environments, no frameworks or extensions have been created to address this need, the programmers currently have to take the challenge of specifying the real-time constraints into their own hands. The Real-Time Java Specification (RTSJ) has been an attempt to address part of these challenges, but it has brought up a number of problems related to the virtual machine and memory models.**

***Index Terms*—C#, real-time systems, thesis proposal, candidacy exam write-up, EDIC, EPFL**

I. Introduction

TIME critical abstractions represent an important research direction for many laboratories around the world. In this context, one important question raised is how would a comprehensive programming model exposed to developers be able to perform the integration of the time-critical with ordinary tasks, in such a way as to avoid, if possible, any negative interference. A pragmatic approach suggests that the language should be extended such that the changes are non-intrusive. The opportunity in protecting libraries and avoiding fundamental changes to the existing semantics resides in the increased chances of adoption of such an extended system, for evident reasons. This yields to a solution where an extended type system would be created on top of the existing one, thus ensuring a perpetual separation of objects in the time-critical context from the ones in the time-oblivion context, with stricter enforcements to the time-critical ones.

Since the real-time context requires a specific scheduler, an important challenge is the integration of such a scheduling mechanism for real-time threads with an existing virtual machine. The intended integration should be as less intrusive for the virtual machine as possible.

Another significant challenge is represented by creating/adjusting a matching ownership type system, intended to be lightweight but expressive enough, and, again, to require as few changes in the existing code as possible. Several existing ownership and heap separation procedures seem to suggest that powerful ownership systems are able to cover most solutions, but they might be over-scaled for our purpose while their abundance in annotations might interfere with legacy code. The goal is to capture the best balance between expressiveness (lightweight) and capacity to address a larger pool of relevant real-time applications.

The integration of the two models, if performed in the above mentioned manner, is able to preserve the advantages offered by the coherence in development tool chains, like development environments, design, debugging, and production tools.

Because of the significant challenges the creation of such a system unveils, and the fact that there are few research results combining hard RT and mainstream programming in one transparent environments, we consider lessons learned from any solutions (even soft real-time) which showed sound practical results. An example is getting help from the experience of the RTSJ [5] (the Real Time Specifications for Java). The real-time capabilities introduced for the Java language shows that the virtual machine is one area where many problems are likely to be faced, since its execution is usually opaque to the programmer, and the control over the memory management (especially through garbage collection) is not deterministic from a programmer's perspective. If for some reason the garbage collection process is able to interfere with a real-time thread (e.g. indirectly through a lock), its operation may take orders of magnitude longer than the timing constraint of the real-time operations, thus breaching the predictability guarantees. Therefore a private memory area for real-time threads is likely to be considered, based on the experience from Reflexes [13], a similar system for Java. The special

Proposal submitted to committee: June 15th, 2010; Candidacy exam date: June 22th, 2010; Candidacy exam committee: Willy Zwaenepoel, Rachid Guerraoui, Viktor Kuncak.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

memory area is partitioned it into at least two regions, a kind of stable, garbage-collection-free area for the real-time tasks, and an additional volatile area to support the interaction between real-time and ordinary threads.

II. Paper I. A non-preemptive scheduling algorithm for soft real-time systems

The research in this paper [7] shows the efficiency of a new developed algorithm for non-preemptive real-time dynamic scheduling. The authors start with choosing to frame the research within the following assumptions: *a non-preemptive environment* on a uniprocessor (which means that newly released threads are not allowed to interrupt any executing thread, hence the elimination of the *priority* property for threads), a *dynamic scheduling* (threads are released at runtime without prior knowledge of their properties, i.e. release time, worst case execution time, etc.), and *both hard and soft real-time constraints* (i.e. threads are allowed to bypass their deadlines by a certain amount).

A. Definitions

The types of real-time deadlines analyzed in this context are *hard* (there is no deadline extension allowed) and *soft* (deadlines may be bypassed by a certain amount).

A *task* is a set of related jobs that jointly provide a function. The three types of tasks with respect to their intervals and deadlines are *periodic* (executed repeatedly at a regular time interval), *aperiodic* (an external event having a hard real time constraint), and *sporadic* (an external event having a soft real time constraint).

Jobs are units of work, that are scheduled and executed by the system. A job has a number of properties in the domain of timings, classified by their absolute or relative times. Absolute times related to a job are the *release time* (moment at which the job is released), *completion time* (moment at which the job is expected to complete under the Worst Case Execution Time (WCET)), and *absolute deadline* (the moment until which the thread needs to guarantee that it has finished its execution). The relative parameters are *Response Time* (the interval from release to completion), *Deadline* (the interval from release to absolute deadline), also referred to as *Feasible Interval*.

The *Load Ratio* is defined as the sum of all expected execution times over the total available times, i.e.

$$\rho = \frac{\sum e_i}{T}$$

and depending on this ratio, we have *light system load* ($\rho < 100\%$) and *heavy load* ($\rho > 100\%$).

The *scheduler* needs to meet some constraints, and it is said to be *feasible* if it has a low miss (jobs started but with deadlines exceeded) and loss (jobs not started ahead of their deadlines) rates. Moreover, a scheduler is *optimal* if besides feasible, it guarantees the production of a schedule in all cases in which such schedule can exist.

B. EDF and improvement solution

One of the most widely-used algorithms for scheduling is the so-called EDF (Earliest Deadline First), whose strategy is to schedule the next job based on the closest approaching deadline. EDF is used for both preemptive and non-preemptive scheduling, and it has been proved that for preemptive environments, it is optimal for all periodic, aperiodic, and sporadic types of tasks, it is also optimal for non-preemptive sporadic tasks, but NP hard for non-preemptive periodic and aperiodic tasks [6]. For light loads, the algorithm still performs well, but for high loads, its performance is poor, because of the **domino effect**: due to the scheduling based on the closest deadlines, once a job misses its deadline (expected under heavy load), the following jobs are also likely to miss their deadlines, too.

Another strategy is Shortest Job First (SJF), which schedules based on the WCET estimations of each job, and always chooses the shortest job. While not an optimal algorithm for all situations, it is still used because in the contexts where short jobs are associated with high priority.

The proposed solution of the authors is called Group-EDF or **gEDF**, is a combination of EDF and SJF. First, it creates groups to comprise jobs with close deadlines. Then, it schedules these groups using EDF, and it schedules the jobs inside a group using SJF. A group therefore contains all the jobs that have close deadlines, starting from the job with the closest deadline. The groups are not equal in length, they have increasing length based on the assumption that two jobs are in the same group if their absolute deadlines span falls within a parameter called Gr:

$$G_i = G_j \Leftrightarrow d_i \leq d_j \leq (d_i + Gr(d_i - t))$$

This is exemplified in Figure 1. Groups are maintained *dynamically*, in the sense that jobs are placed in groups upon the completion of an old job; because groups span are relative to the next groups change with the progress of the system. The algorithm maintains a queue sorted by deadlines. The progress is the following:

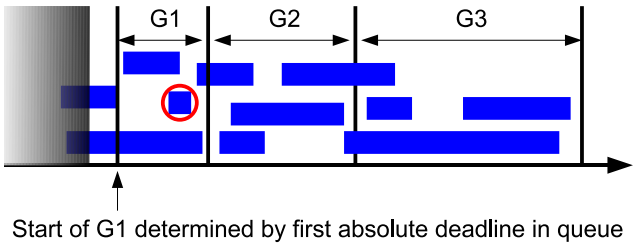

Fig. 1. Groups creation

- upon job j release: if $d_j > t$ then insert job in queue by deadline
- upon completion of an old job: if queue not empty, search within group G1 for the shortest job, dequeue it and run it

The algorithm has the same complexity as EDF of $O(n)$, but with a little larger execution time due to the search performed upon completion of a job. Because of the use of SJF, it will tend to favor somehow the shorter jobs, and it might not

guarantee fairness (same average allocation for short and long jobs).

C. Evaluation

The authors perform several experiments, starting with adjustments to the Gr parameter. Indeed, if this parameter is set to 100%, it means that gEDF is exactly EDF, while if it is set to 0%, it means that gEDF is exactly SJF. The authors conduct their experiments with the following properties:

- Types of measurements: *success ratios* (% of completed jobs ahead of deadline), *success ratios improvements* (e.g. gEDF vs. EDF) and average response times, everything for loads from 0% to 300%
- Parameters varied:
  - *deadline tolerance Tr*, ability to exceed deadline by this value)
  - *deadline tightness* μ_D, how tight deadlines are with respect to the execution time, $D = \mu_D * e$)
  - *classes of expected execution time* μ_e
  - *group range parameter* Gr
- algorithms compared: gEDF vs. EDF, gEDF vs. Guarantee, Best Effort and EDF algorithms, with fixed parameters

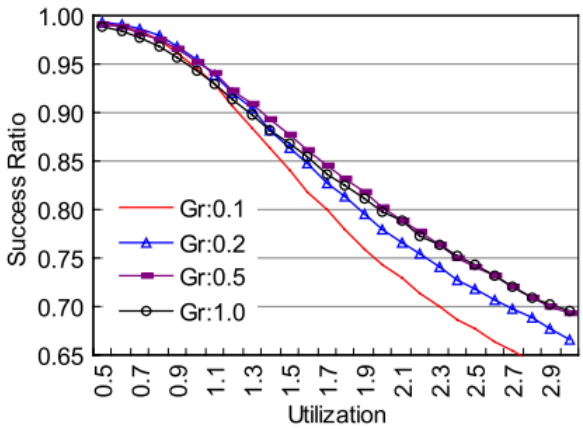

Fig. 2. Success ratio for various Gr values

The first experiment is targeted to finding the best value for the Gr parameter. Figure 2 shows the success ratio of the algorithm, with respect to the load, for various values of the Gr parameter. The result shows that the optimal value for Gr is about 40%, which will be the value used in all following experiments.

The following experiments would vary one of the above parameters for each trial. The default values for the trials will be $Gr = 40\%, Tr = 20\%, \mu_e = 40, \mu_d = 5$. The experiments that vary the deadline tolerance ($Tr = 0\%, 50\%, 100\%$) and deadline tightness ($\mu_d = 1, 2, 5, 10, 15$) are presented in Figures 3 and 4 and show that in all cases gEDF improves over EDF, and this improvement increases with the tolerance.

The next experiment shows that, as expected, gEDF tends to favor short jobs. The expected execution time is varied, in 4 trials, from mostly longer jobs (Distribution 1) to shorter

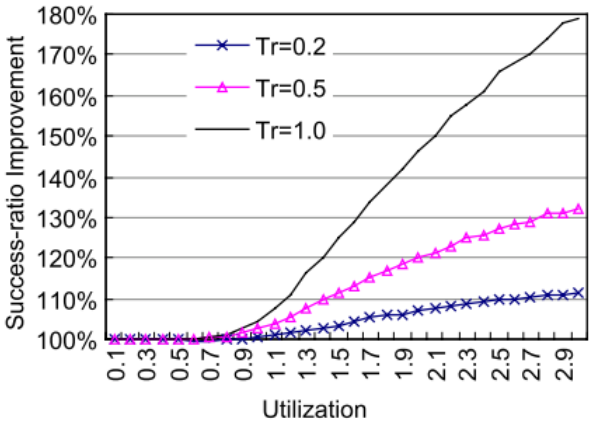

Fig. 3. Deadline Tolerance improvement

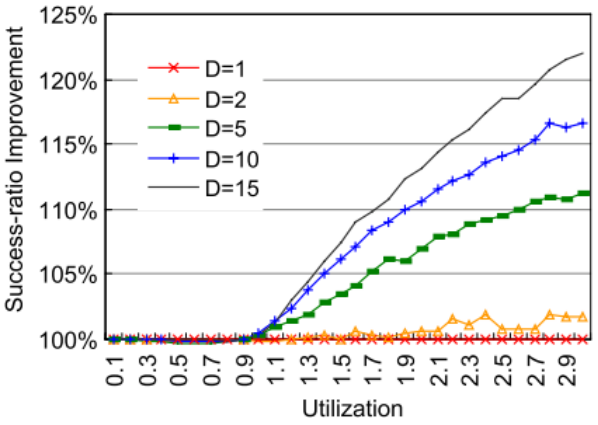

Fig. 4. Deadline Tightness improvement

jobs (Distribution 4). Figure 5 shows the improvement ratio of gEDF over EDF for these distributions.

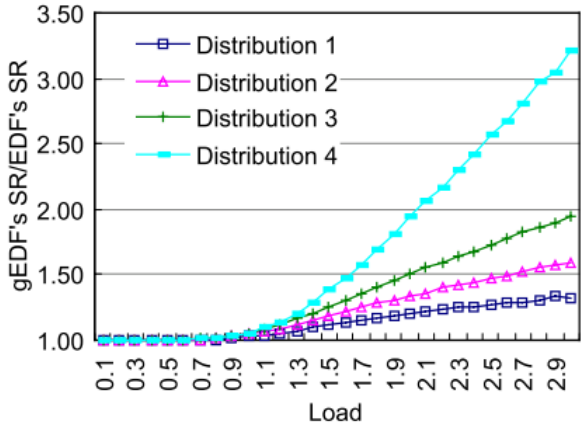

Fig. 5. Improvement with respect to job durations

The last experiment presents the comparison between the most important techniques available for non-preemptive real-time constraints, i.e. gEDF vs EDF, Best Effort and Guarantee.

Best effort [8] is an efficient algorithm with selection based on a *value density* parameter, $\rho = \frac{V}{C}$ where V is the value of

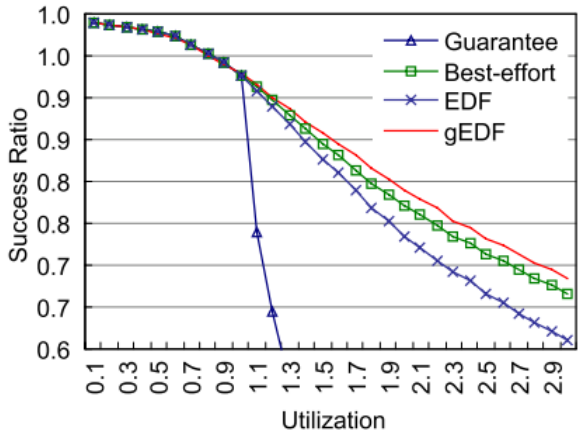

Fig. 6. Success Ratio of the four algorithms

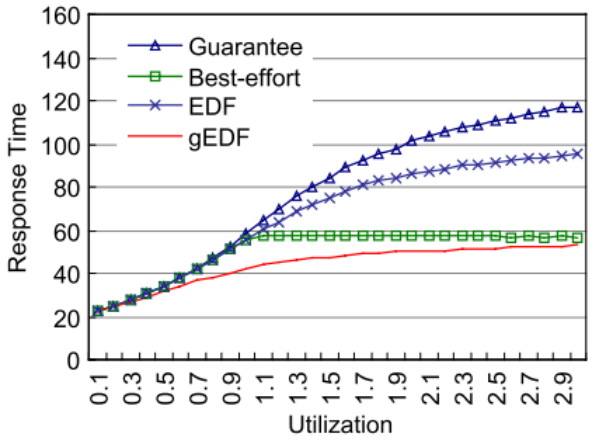

Fig. 7. Response Time of the four algorithms

the job (or the priority), and C is the expected WCET. The experiments use a constant V for all jobs. The Guarantee [2] scheme uses an acceptance test to queue the jobs, therefore infeasible jobs are not scheduled at all. The jobs that are feasible, however, are scheduled using a First Come First Served policy. This scheduling is infeasible for overload, which is reflected in the success ratio. The graphics presented in Figures 6 and 7 show the compared success ratios and response times for these four algorithms.

As a conclusion, this paper presents an improved scheduling solution, for a non-preemptive environment, with advantages over other solutions for hard real-time deadlines and light loads, and with huge improvements for soft real-time deadlines and high loads. This technique is appealing for its simplicity, for its small complexity that does not exceed the one of the state of the art EDF, and for the ability to deal very well with heavy loaded environments.

III. Paper II. Stretching Transactional Memory

Software Transactional Memory STM is a software paradigm that applies transactional databases constraints to memory, to be able to obtain a shared memory concurrency control system, a mechanism to simplify, for example, parallel programming. In a transactional memory, accesses have to be *atomic*, i.e. they either execute or not (commit or abort), and leave the system in a *consistent state* at all times (i.e. no access sees an inconsistent state at any time).

A successful implementation of such a STM is SwissTM [4]. Some other implementation with various design choices are TL2 [3], TinySTM [11] or RSTM.

A. Transactional Memory

The most frequent issues found in transactional memory are conflicts over the same memory location (two threads trying to access the same area at the same time). The first challenge here is *how to detect* when this contention happens, the second one is *how to proceed* in this case. Also, a significant decision point regards the level of granularity of the memory that needs to be locked, which can be just a word, or up to fields or even whole objects. Transactions accessing the shared memory are typically Reads (R) and Writes (W), and the conflicts appear between R/W and W/W transactions.

For the first issue, there are typically two approaches:

- lazy: the detection is made at commit time (e.g. TL2). This typically favors short transactions, and is inefficient for long transactions that might abort right in the end, wasting computational time
- eager: the detection is made at encounter time (e.g. TinySTM), which triggers the contention manager early, estimating in advance that long transaction might fail, but not favoring read-write conflicts which typically succeed without aborts if allowed to proceed

Along with the detection moment, it is also important to establish who is responsible for the detection, which translates to the way the reads are:

- visible: the read transaction is visible to other transactions accessing the same memory
- invisible: the read transaction is not visible, therefore the read transaction itself has to deal with conflict detection

The second element, the contention manager, has to provide logic for deciding which thread is aborted, and which one is granted access to the memory area. The two transactions are called the *attacker* and the *victim*, usually the victim being the first one to get access to the shared resource. Possible operations are:

- timid: abort attacker (favors short transactions)
- Polka: priority abort, based on number of objects accessed by that specific transaction
- Greedy: priority abort, based on transaction start timestamp (favors long transactions)

B. Operation

SwissTM tries to improve the overall outcome for the whole wide-spectrum of possible transactions, from short to long, and from simple to complex workloads. For this, the design choice is to use *invisible reads*, a combination of *lazy detection for R/W* and *eager detection for W/W* conflicts, and a combination of *timid contention management for short transactions* and *greedy for longer ones*. The distinction of the latter two is

made by the number of accesses W_n, if this number exceeds 10. The greedy parameter is a global timestamp $greedy - ts$. As an optimization, the transactions not reaching W_n do not access this timestamp to reduce contention on it.

The setup of SwissTM consists of a global commit timestamp used for versioning and state synchronizations, which is incremented upon every successful commit. Next, each memory location contains two locks, one for reads and the other for writes, that point to the transaction currently holding that lock, containing a timestamp field for versioning. In order to prevent inconsistencies, the $r - lock$ is acquired at commit time, then released after commit; $w - lock$s are acquired eagerly. The transactions have read and write logs attached, which are used for all operations performed by the transaction for each memory location a transaction accesses. The operations are performed on these logs, until the commit time when the updates are transferred to the memory. The transactions are also equipped with a validation timestamp, used to check for consistency of

The contention manager is an external part of the system, called every time a conflict is detected. The structure of SwissTM is presented in Figure 8.

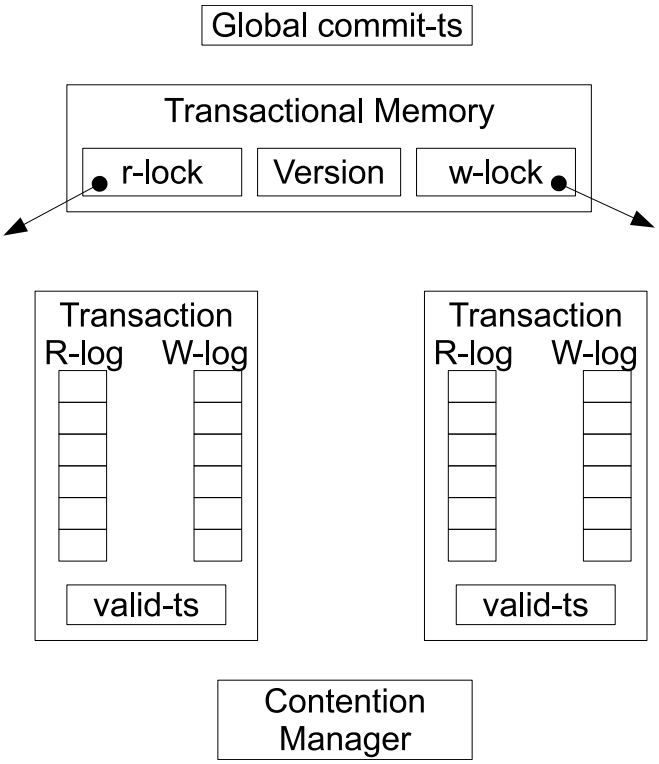

Fig. 8. SwissTM Design

The operations performed by the system are the following:

- Read(T,m): a transaction tries to read a location. If T holds the $w - lock$ of m, it will return the value from its $w - log$. Otherwise it will repeat two consecutive reads until they present the same $r - lock$. If the version of m is lower than T's timestamp, the read value is returned, otherwise a validation is triggered, its success yielding the return of the value, while its fail will make the transaction rollback
- Write(T,m): if T holds the $w-lock$ it writes to its $w-log$. If not, T tries to set the lock to its log using compare-and-swap, an assumed atomic operation. If this fails, a conflict is in place, so the contention manager is called. The transaction also checks the $r - lock$ to keep opacity, and revalidates its read locks
- Validate(T): check whether all read log entries have the same version as their references, for all memory locations not locked by T
- Rollback(T): clear all write locks
- Commit(T): if read-only, just return, the read log is consistent. Otherwise, acquire locks for all $r - lock$s and $w-locks$, rollback if validation fails, or else update write locks and release all locks

The contention manager chooses the timid or greedy approaches based on the number of objects write-accessed by the transaction. If this number exceeds 10, every new access will increment the global greedy timestamp which will be used in the greedy approach.

C. Results

SwissTM's evaluation is done using several Benchmarks, as micro-benchmarks (red-black trees), STMBench7 (realistic, complex, and object-oriented applications), STAMP (many selectable programs and workloads, few long transactions) and LeeTM (large, realistic workloads). STMBench7 is a state of the art benchmark to test the intended functionality of SwissTM for long and complex transactions. When using this benchmark, SwissTM outperforms other STM's in read-dominated workloads, and also improves upon them in write-dominated workloads, as shown in Figures 9 and 10.

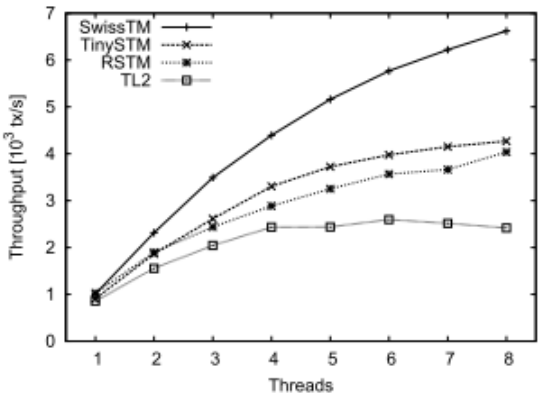

Fig. 9. Read-dominated workload (90% reads)

Software Transactional Memory can be an efficient paradigm to use in the context of communicating between tasks of different frequencies or priorities. Therefore such transactional objects could be the solution to the problem addressed by possible inconsistencies in resource sharing, able to waive issues like blocking or priority inversions.

IV. Paper III. Preemptible Atomic Regions for Real Time Java

The authors present a new abstraction called Preemptible Atomic Regions [9] (PARs), which is an improved concur-

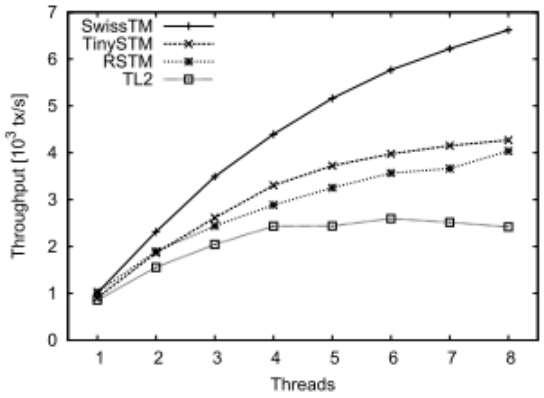

Fig. 10. Write-dominated workload (90% writes)

rency control mechanism, targeted at improving over mutual exclusion in a few directions, namely *strong correctness guarantees* i.e. all atomic operations will not suffer from threads interferences, or *high priority tasks preemption* in order to reduce blocking times.

Concurrency control still suffers from old behaviors and approaches, critical sections are still kept to sizes as small as possible, which might not scale well with today's requirements. The PARs are a restricted form of STM, an alternative to monitors, based on a sequence of instructions that have atomic execution guarantee. This means the effects of the PAR are completely rolled back when a higher priority task preempts a lower one.

A. Real Time Java

The RTSJ allows the co-existence of real-time and normal threads. The integration is not seamless, but it does not require any changes to the development chain (compiler, IDEs, etc). The key distinction of the two type of systems is in the way the memory management is done: the real time threads operate in a private memory region which is exempted from garbage collection operations, the deallocations being made in constant time. The normal threads still use the heap and the garbage collector. The concurrency control is implemented using locks and priority inversion is typically solved using priority inheritance, or with optional priority ceiling.

A significant issue is the computation of the WCET, which is not trivial due to the need to estimate the longest critical sections, the priority inversion logic which requires runtime effort, and the possible blocks of a real-time thread on either normal threads, or indirectly (through a normal real-time threads) on the garbage collector.

B. PARs Description

Since the operations of threads inside a PAR are undoable, no external threads are able to see the effects of these operations, therefore atomic methods are easily aborted when a higher priority thread is released, and the blocking time is reduced to the duration of the abort. Also, because only one PAR can be active at a certain point, there is no need for more than a single global undo log. PARs offer support for nesting, i.e. executing a new internal PAR region within an existing one, but the granularity control is specified such that upon an abort, all nested PARs including the outermost will be rolled back.

Compared to lock-based implementations (like monitors), PAR's have a number of advantages:

- less lock acquisition overhead: PARs only require keeping a pointer to the current thread, and the log reset on exit is done through a single pointer change; monitors need to maintain several locking queues and perform several allocations
- less nesting overhead: nested PARs entrances and exists can be ignored because PARs can only conflict with other PARs
- less context switching overhead: lock based implementations need significantly more context switches in the presence of many threads with different priorities

but also a few disadvantages, among which

- PARs need to log every (write) operation performed
- rollbacks are more expensive, if the environment is write-dominated
- PARs are only applicable to undo-able operations, hence no I/O operations can be executed inside a PAR
- no blocking operations like `wait()` or `notify()` can be executed in PARs

C. Implementation

Since PARs are a concurrency control protocol, they need to implement the transactional abstractions of *read*, *write*, *commit* and *abort*. Reads are safe and can be performed directly, because only one PAR runs at a time. Writes require prior pushing of the current memory location and value to the undo buffer, then write to memory. Abort goes through the undo buffer and restores the values. The abort costs $O(n)$ where n is the length of the log. Commit does nothing because writes are performed directly to the memory. A complete conflict (or contention) manager is not required for PARs, because the only decisions are due upon releasing a new thread with a higher priority. Therefore only a simple version is implemented. If the other thread is in a PAR, the new one is released and will trigger the abort. If the other thread is already aborting, the new one has to wait until the completion of that abort. The methods subject to PARs are annotated by the programmer with `@PAR`. The bytecode needs to be rewritten with a transformation of methods, as in Figure 11, from method `f()` to `f$()`.

Concretely, the methods annotated are captured by reflection, and the $ methods are created along. The system is integrated into the OVM real-time Java virtual machine. OVM employs its own preemptive scheduling system, that has been adapted to support PARs. Upon a context switch, the simplified contention manager is invoked. The authors deal with possible reflective methods invocations (RMI) by logging them and RMI calls are redirected to these logged methods.

The memory management is done via a system-wide undo log, preallocated statically in immortal memory, not subject to garbage collection, and having a fixed size, wide enough to

```
void f() {
   while (true) {
      try {
         try {
            PAR.start();
            f$();
         } finally { PAR.commit();
            PAR.exit(); }
      } catch (AbortedFault _) {
         continue; }
      break;
   }
}
```

Fig. 11. Code transformation at runtime

accommodate typical log sizes. To address an issue of memory leak in the case of many repeated aborts, the effects of all memory allocations have to be undone. A solution is to record all the allocation pointers upon start, and restore the allocation pointers upon abort.

D. Evaluation

Evaluation has been performed for a priority preemptive scheduler, measuring the response time. There are n tasks scheduled using a rate monotonic scheme, with one critical section per job.

The results of PAR trials versus the Synchronized Lock-based solution, for a HashMap micro-benchmark, is presented in Figure 12. There are two threads, a hight priority and a low priority one. The low one executes critical sections with read, insert, delete operations. The high one also executes a similar number of operations, periodically. The upper plot on the graph shows the response time for the synchronized version, which uses priority inheritance; the lower one shows the PAR implementation.

Results show that PARs are better in response time (i.e. one abort including rollback of all low priority threads writes is faster than the two context switches) and also, perhaps even more important for real-time threads, the PAR implementation is *more predictable*, response times being almost constant.

The PARs represent a good abstraction for controlling concurrency for real-time programs, having a stronger correctness guarantee than normal locks. Experiments also prove that PARs have smaller overheads and experience less jitter. Therefore PARs may prove to be a good solution for scheduling hard real time threads on a single processor.

V. Research Proposal
Towards Integrating Hard Real-Time Capabilities with Time-Oblivious Applications in C#

As the title suggests, the research will focus with addressing the creation of the types and semantics of the RealSharp abstraction, while tackling the foreseeable issues presented in the introduction to overcome them.

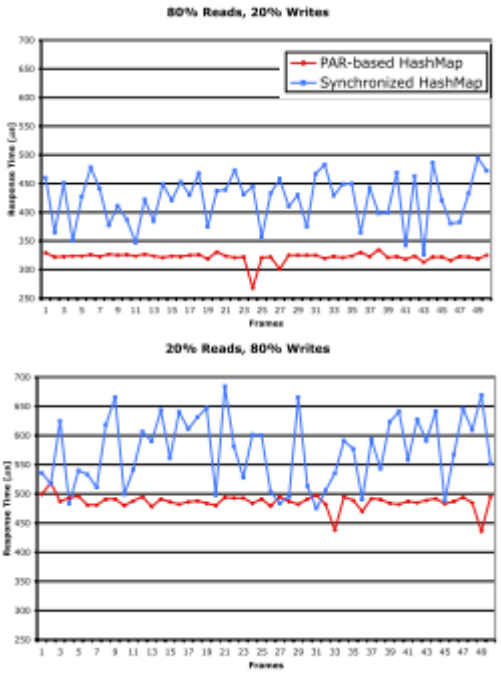

Fig. 12. Response time for PAR vs Synchronized. Lower is better. Upper figure: read-dominated. Lower figure: write-dominated.

A. Experience

During the first semester within EDIC, I worked on a project with prof. Viktor Kuncak on Null-Reference Analysis for the `scala` language, where I developed skills in working with types and compilers insights. In the creation of real-time specifications for C#, building new types upon the existing type system will benefit from this experience, as this operation has to be done with caution.

During the second semester, I worked on a project with prof. Rachid Guerraoui, on attaching adjustable deadline logic to a transactional memory on C#. We worked on the first transactional memory for C#, SXM [10] from Microsoft Research, and then on SwissTM for C#, whose port was in progress at that time.

Specifically, we tried to address the situations when, for some reason, a transaction is repeatedly unable to acquire a lock on some location, which translates into significant loss of computational time, or at least a disorganization of the transactional logic. Another aspect tackled is a time-bounded execution strategy on the transactions, even if they were able to acquire locks and progress. The solution was to attach deadlines, therefore a transaction that is unable to progress or finish its job by its due deadline, would be aborted and terminated, with additional selectable logic (retries, deadline extension or restraining, depending on the context). To our best knowledge, attaching deadlines to accessors of transactional memory is a new research topic, though our contribution may be significantly improved in a number of ways, one of the most appealing being the creation of a scheduling logic with deadlines rather than just attaching a deadline to every thread.

The C# implementation involved code instrumentation using Reflection, the runtime C# paradigms for dynamically ac-

cessing executing code within an executable assembly, which required a significant amount of Intermediate Language (IL) code manipulation. During this phase I developed skills of working with IL, with reflection (`OpCodes`), and improved my C# knowledge overall. Plus I got more acquainted with the atomic operation mechanisms implementations in the .net framework, notably `Interlocked`, with the use of `ThreadStatic` fields and operations, with `Attributes` for runtime class and modules custom treatments, etc.

B. Proposed solutions

A fair amount of ideas come from *Singularity*, which is a Microsoft Research experimental operating system, written mostly in managed code, namely using Sing#, an evolution of Spec#, which integrates C# with contract-based programming. The code is compiled into Singularity using an experimental compiler, called *Bartok*, which allows the implementations of the garbage collector and other components to be chosen at runtime on a per-application basis, optimized for specific usage. One solution this existing work may provide is within the creation of the private memory areas, using Singularity's per-process GC which could be invoked upon stable memory at idle times of the RealSharp tasks.

Given the experience in the various Java implementations of real-time abstractions, proposed in RTSJ [5], StreamFlex [12], Reflexes [13], or Flexible Task Graphs [1], a direction in this research will be to establish whether the C# type system and framework differences from Java can be used for the proposed approach, and whether the intended abstractions are able to guarantee predictability and also performance of the running system.

Regarding the communication among RealSharp tasks, studies will have to confirm whether an implementation based on Singularity's channels with messages allocated in a restricted inter-process shared memory region, or the encapsulation approach proposed in StreamFlex [12], which would allow passings of references to arbitrary immutable data structures. More precisely, the StreamFlex environment employs restrictions to primitives and arrays for transferring data types along channels to ensure safety, but a closer investigation seems to suggest that restrictions on primitives can be further lifted if a static check is able to ensure immutability of references for this data.

Another direction of research will be to address calls exiting from the Virtual Machine, like I/O calls and native calls, which were characterized in the PARs description as unsupported because they cannot be undone. Their unpredictable nature (in terms of WCET) forced the implementation of StreamFlex to exclude them from interacting with real-time threads, therefore leaving the programmer to deal with this responsibility. It will be interesting to see whether these conservative constraints will be able to be lifted or reduced.

The implementation will therefore cover first the programming abstractions, where the transactional objects used for the synchronization of the shared data channels will also be included. Second, an ownership type will be created, along with type checker extensions, to be passed to the Bartok compiler for type rules enforcing on the application code. Then, the Bartok runtime extensions that would support memory-area switching, class based allocation switching, or transaction logging should be approached.

The evaluation of the system is foreseen to include studies over the programming model and its implementation. Since the key goal for the real-time area is predictability, microbenchmarks can be created to ensure the proper intended behavior. However, the key aspect being in the integration with normal tasks, the evaluation of the overall system (using real-life scenarios) seems to be more useful for the assessment of the approach.

We expect to obtain a simple, non-intrusive extension of the C# language, that would support both real-time threads and normal, time-oblivious operations in the same environment, without requiring too much intrusion to the existing framework, and thus giving opportunity to programmers to leverage their experience on the same abstractions and tools offered by the C# environments.

REFERENCES

[1] Joshua Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible task graphs: a unified restricted thread programming model for java. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 1–11, New York, NY, USA, 2008. ACM.

[2] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. *Real-Time Systems Symposium, IEEE International*, 0:90, 1995.

[3] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[4] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM.

[5] Real-Time Specifications for Java. http://www.jcp.org/en/jsr/detail?id=1.

[6] Kevin Jeffay and Donald F. Stanat. On non-preemptive scheduling of periodic and sporadic tasks. pages 129–139, 1991.

[7] Wenming Li, Krishna Kavi, and Robert Akl. A non-preemptive scheduling algorithm for soft real-time systems. *Comput. Electr. Eng.*, 33(1):12–29, 2007.

[8] Carey Douglass Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Pittsburgh, PA, USA, 1986.

[9] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Microsoft Research. SXM Software Transactional Memory. http://research.microsoft.com/en-us/downloads/FBE1CF9A-C6AC-4BBB-B5E9-D1FDA49ECAD9/default.aspx.

[11] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[12] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 211–228, New York, NY, USA, 2007. ACM.

[13] Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: abstractions for highly responsive systems. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 191–201, New York, NY, USA, 2007. ACM.