Using Transactional Memory to Synchronize an Adaptive Garbage Collector in Real-Time Java

M. Teresa Higuera-Toledano
Facultad Informática, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Saín
Email: mthiguer@dacya.ucm.es

***Abstract*— Currently multicore systems start to be used even in low-end devices such as embedded systems controllers requiring real-time guarantees. Hardware Transactional Memory (TM) is a new synchronization paradigm for this architecture, allowing problems of lock-based methods and making easer programming. We propose to use TM to synchronize the concurrent/parallel execution of the Garbage Collector (GC) and Java applications, within the context of an embedded real-time environment. GC gives robust programming, and performance advantages. Since in the context of real-time systems, a collector must guarantee a worst-case in it performance, the worst-case time for a committed transaction must be limited; we achieve this by limiting its size (i.e., the size of the explored object).**

Keywords-component; Memory Management, Real-Time Java, Garbage Collection, Synchronization Paradigms, Transactional Memory.

I. Introduction

Real-time and embedded applications cover an extremely wide variety of domains, each with stringent requirements. This diversity of requirements includes highly precise timing characteristics, small memory footprints, flexible sensor and actuator interfaces, and robust safety characteristics. Nowadays, research in Computer architecture focuses on *multicore* architecture (i.e., two o more processor core reside in a single chip sharing memory). Each core maintains a local cache, which requires maintain the coherence for shared dates by using special protocols in the cache controllers. Programming such systems in safety and efficient way requires a new paradigm of synchronization. Concurrent and parallel programs are typically synchronized by locks, which use is not trivial.

Transactional Memory (TM) is an alternative to the lock-based synchronization [10]. We can see a transaction as a piece of code that is running a series of reads and writes to shared memory. These reads and writes occur at a time (i.e., such actions are indivisible) because intermediate values in the transaction are not visible. TM is an attractive concept expressing parallelism, which with the advent of multi-core systems, becomes more interesting because avoids in an elegant way the problems of lock-based synchronization. Currently this is one of the hottest topics research in several areas: (e.g., programming languages, computer architecture, and parallel programming).

Garbage Collection (GC) automates memory reclamation, making programs easier to write and correct, and also more efficient and robust, while by replacing locks with transactions we can make shared-memory parallel programs easier to write and correct, and also more efficient and robust. This perspective is showed by Grossman [7] [12] as an original analogy:

"Transactional memory (TM) is to shared-memory concurrency as garbage collection (GC) is to memory management."

Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes along with overhead regarding both execution time and memory consumption, which makes (implicit) garbage collection poorly suited for small-sized embedded real-time systems.

Real-time garbage collection must assure memory availability for newly created objects without interfering with the real-time constraints. Most garbage collectors do not make real-time guarantees. Providing such guarantees requires take into account the worst-case cost, which can incur substantial extra cost in the expected case.

The problem here is that the GC must progress with its collection while the program (*mutator*) must access arbitrarily on the reachable objects while the collector is analyzing it. Some implementations of transactional memory provides also worst-case, but incur substantial extra cost in the expected case execution time; again the problem is the transaction progress while a thread could be suspended after having accessed any of the objects the transaction is accessing. Efficient GC/TM approaches can involve code generation, static analysis, run-time systems, and hardware (e.g., [22], [25]).

The Java environment has become an attractive choice for real-time embedded systems because of its safety, productivity, relatively low maintenance costs, as well as the wide availability of well trained developers. However, Java is unsuitable for developing real-time embedded systems, because its under-specification of thread scheduling and the unpredictability of garbage collection. To address these problems, some significant extensions to Java have been introduced, such as the Real-Time Specification for Java (RTSJ) [26].

TM seems a promising approach for dynamic data structures and applications with independent threads; we found interesting to investigate how TM can synchronize the cooperation relationship in the parallel execution of the GC and the application. Within the context of real-time Java, we propose to use the TM paradigm to synchronize a real-time GC and the application; our solution comes along with the integration of existing solutions. Regarding the use of TM to synchronize the application and the GC, we chose the Dijkstra tri-color concurrent algorithm [6]. This is a graph-based algorithm allowing the concurrent execution of the GC and the applications threads.

A. Related Work

The literature provides us some approaches to transactional collectors. The work presented in [35] gives an approach to a transactional GC algorithm, which safely collect garbage while maintaining consistency in a concurrent transactional storage context environment. It uses train-based generational algorithm. Another examples intersecting GC with transactions are [Amsaleg] which study the interaction of a GC in the setting of a OODB at a detailed level. In [18] a semi-space copying collector is adapted to collect a transactional persistent heap. However, these three GC collects spaces uses by applications using transactions such as BD, while our proposed approach use transactional memory as synchronize mechanism among the GC and the application (i.e., [35]).

While Software-based TM has been studied for different purposes (e.g., in data bases or multi-cores systems) their use in real-time systems is still practically unexplored. However, we can found an interesting Software-based TM solution in [Sarni], which studies the WCET under a real-time operating system called LITMUS.

Hardware-based TM solutions have been widely studied; the solution presented in [24] deals with time-based TM, using time to reason about the data consistency and the order of transactions committed. However, this considers only throughput, not WCET such as real-time systems requires. A different approach is such multiprocessors-based [32], where a time-predictable TM solution has been presented as a new concurrency abstraction for shared-memory. This solution integrates the number of retries in a transaction into the WCET analysis and targets the Java environment. Choosing Java presents great advantages in embedded systems over low-level languages.

Within the Java environment, the paper [30] analyses the difficulties of a parallel mark-and-sweep GC. The GC is one of the largest sources of unpredictability in Java regarding its use for real-time systems. The original solution of the Jamaica VM [31] bounds the GC unit works by dividing the objects in blocks of the same size.

The adaptive GC presented in [36] changes the GC priority within a Java real-time environment, in order to achieve the optimal balance between the application throughput and the GC pause times. However this solution can be only used in soft-real-time systems.

The work presented in [15] deals with the parallelization of the Dijkstra SSSP algorithm on multicore platforms. This is a graph algorithm used to compute single source shortest paths, which is hard to parallelice. The Simulation results, presented in [16], demonstrate that traditional locks and barriers have a disappointing performance, which can be higher improved by using TM techniques to efficiently synchronize concurrent accesses of threads to shared data structures.

B. Paper Organization

The rest of this paper is organized as follows: We begin by providing a review of transactional memory concepts (Section 2). Next, we present a review of garbage collection techniques (Section 3). Following, we introduce the main issues in the design of a TM-based real-time GC, analyzing the problems that this solution presents (Section 4). And finally a summary of our contribution concludes this paper (Section 5).

II. TRANSACTIONAL MEMORY TECHNIQUES

Transactional Memory techniques are associated with databases for controlling access to shared memory in concurrent computing. It functions as an alternative to lock-based synchronization. A transaction is a set of operations on data that are guaranteed 100%. A transaction is a piece of code that executes a series of reads and writes to shared memory as an atomic operation; intermediate states are not visible.

Changes made to a transaction are validated and, if the validation succeeds, the changes are confirmed. The transaction can also be avoided at any time, causing that all changes get cleaned. Whether changes are validated and made permanent, the transaction *commits*, and whether a transaction cannot be committed due to conflicting changes, it is *aborted* and re-executed from the beginning until it commits. This process repeats the transaction until it is achieved.

The main attraction of this technology is the ability of good performance with parallel code using coarse grain transactions.

A. *Transactional Memory Synchronization*

TM techniques are very optimistic: a thread completes modifications to shared memory without regard for what other threads might be doing, because the onus is placed on the reader, who verifies that other threads have not concurrently change the memory that it accessed, instead of placing it on the writer to make sure it does not adversely affect other operations in progress. A TM system must meet the following properties at all times [9]:

- **Atomicity:** the transaction is executed completely or not run at all.
- **Isolation:** the interim statements traversed by partial transaction (i.e., not completed) should remain hidden to other transactions.

An approximation to TM is the Linux Read-Copy-Update (RCU) technique, which makes a copy of a data structure that a process modify, leaving the original intact, while at the same time other processes can read the original data structure without suffering any kind of waiting. Once modified the structure, it is updated to new readers, while old readers continue reading the old data structure. Once there are any process reading the old data structure, it is deleted from memory. This technique is one of the keys allowing Linux to run on multiprocessor systems.

B. *Programming using Transactional Memory*

The lock-based mechanism can be used to *i)* provide serialized access to shared data, *ii)* create barriers or *rendevouz* points, or *iii)* ensure temporal ordering. The main difference between transactional memory and lock-based critical regions relies in that locks ensure atomicity by allowing the progress of only a thread, while TM allows the progress of all the threads. In addition to their performance benefits, TM techniques makes easy the conceptual understanding of parallel programming integrating well with existing high-level abstractions such as objects and modules [9]. Lock-based programming is error-prone requiring the programmer to think about overlapping operations and partial operations in different sections of the program code, that are any times unrelated among them[1].

The concept of a memory transaction is simpler than the lock concept; developers express what regions of the code must be executed in *mutex*, rather than having to specify how to reach the atomicity using critical regions or other explicit synchronization constructs. For each transaction, the start and end of each transaction can be delimited through blocks **atomic**, which appear explicitly within the program code. Deadlock is prevented or handled by an external transaction manager.

TM techniques combine an intuitive interface and an efficient implementation, where programming is similar to the use of coarse grain locks while achieved scalability and performance of the fine-grain locks. Coarse-grain looking allows more consistency at the cost of less concurrency, while fine-grain looking provides more concurrency, but less consistency.

C. *TM Implementation Issues*

The idea to adapt the transaction concept used in database as synchronization mechanism integrating it in programming languages was propose in [20]. The TM programming model can significantly reduce the difficulty of writing correct concurrent and parallel programs, and offers an alternative way to simple synchronization and a more efficient solution than locks because TM can be implemented in hardware. There are implementations of transactional memory *i)* software, *ii)* hardware, and *iii)* hybrid.

Software-based Transactional Memory. There are two underlying implementation method for software-based TM (STM): lock-based that provides mutual exclusion to some stages of the transaction, and obstruction-free that guarantees progress for every process having not contention (i.e., a process that is preempted, delayed or crashed does not affect other thread to progress).

In general, STM cannot perform any operation that cannot be undone, (i.e., I/O operations), and are limited to static transactions (i.e., the size of the data that compounds the transaction is known in advance) [27]. However, such limitations can be overcome; the irreversible operations can be achieved by using buffers that queue up these operations, performing them later, outside the transaction, which can be enforced at compile time, such as in [10]. Also, the static limitation has been solved within the Java environment in [29] by providing software allowing dynamic transactional memory.

Hardware-based Transactional Memory. The idea of supporting hardware for memory transactions (HTM) was introduced in 1986 [17]. This idea has been already devised at the beginning of the 90 [14], and it has gained prominence in 2004 with the work presented in [21]. These implementations significantly increase performance by modifying critical elements (e.g., caches coherence protocol) to ensure the properties of atomicity and isolation.

Hybrid Transactional Memory. These systems implement in hardware certain mechanisms to reduce the overhead of software transactions (e.g., [13], [28]) providing a flexible system, to the time to get a simple virtualization. As an example, we can use hash hardware to accelerate the

[1] The programmer must use a locking policy to prevent *deadlock* ensuring the application progress, which are often informally enforced and fallible, and when a fail arises it is insidiously difficult to reproduce and debug.

detection of conflict, while all the other transactional functionalities are implemented in software.

D. Key Features in Transactional Memory

The basic mechanisms required are *i)* coherence and consistency, *ii)* conflict detection, *iii)* conflict resolution, and *iv)* transaction semantics. Various policies can be applied to each of these four mechanisms, which compound the design space for HTM.

Coherence and Consistency. This mechanism, also called Version Management, handles the two simultaneous versions of data storage that are changed for each transaction: the new value will be visible if the transaction *commit* while the old value is maintained only where the transaction is *aborted*. Only one the two versions can be saved to the memory address of the data *in-situ*. Depending on what value is stored in the field and what aside, the version management policy is called: *i) anxious* or *eager*, whether the new value is stored in-situ, and *ii) lazy* whether the old value remains in-situ.

The HTMs with anxious policy keep the old values within an *undo-log*, which is used to restore the old values only whether the transaction is aborted. While the HTMs with lazy policy use a writing buffer or the own cache to temporarily keep the speculative state, and the new versions are written to memory (i.e. whether the transaction is committed) or discarded (i.e. whether aborting the transaction).

An anxious version management makes the commits faster than abortion because the new values are already in place of corresponding memory and we only need to discard the log. The *Unbounded Transactional Memory* (UTM) [1] and *Log-based Transactional Memory* (LogTM) [28] are two different solutions presenting anxious updates policy.

On the other hand, a lazy policy makes easier to remove the updates and to abort the transaction because the memory values remain the same at the transaction start; then we have only to empty the speculative buffer. In general, we though hopefully that commits be more frequent than aborts, using anxious version management (e.g., [8] [5]). Other proposals using this policy also support unlimited size transactions such as *Large Transactional Memory* (LTM) [1] and *Virtual Transactional Memory* (VTM) [23] which use data structures in virtual memory, leading to a significant increase in the complexity of these systems.

Both STMs and HTMs use the concept of eager/lazy versioning. There are also some hybrid approaches [13] using one approach for read-set and another approach for write-set.

Conflict Detection: A transaction conflict occurs whether the data written by a transaction overlaps the data set read or written by another concurrent transaction. This occurs when two transactions access the same memory address and one or more of these accesses are a write operation. To support conflict detection, we must follow the data read and written by each transaction track, which can be performed with block or word granularity. In the first case, only two bits per block (i.e., R and W) are necessary, but may appear false conflicts when concurrent transactions access different words in a block. Accounting sets R and W per word granularity avoids this situation, but requires 2 bits per word to save this information.

Conflict detection policies vary in function of when examines information sets R and W. Many proposals increase the cache memory with two bits, R and W, to indicate whether the block has been read or written by the current transaction. Depending on the cache coherency protocol used to detect conflicts, we have two policies: *pessimistic* (also called *anxious*) and *optimistic* (also called *lazy*). Pessimistic polices detect conflicts when making the memory reference (e.g., LTM/UTM [1], VTM [23] and LogTM [28]), while optimistic polices delay detection until the first conflicting transaction attempts to commit (e.g., [5]).

Pessimistic conflict detection can improve performance because it can avoid abortions. While an optimistic approach can mitigate the impact of chained conflicts and enables deployment of conflict checking batch.

1) Conflict Resolution: The conflict resolution policies are independent of the time that the conflict is detected. Depending on the scenario, we can choose to stop the petitioner (i.e., the committer) or to abort to other threads. Generally, the HTM optimistic detection policy resolves the conflict leaving the transaction that attempts to commit to win the conflict, aborting to rest (e.g., [8] [5]).

2) Transactional semantics: It is obvious that transactions must be atomic each other. However, the relationship among a transaction and the non-transactional code can result in two different models. Regarding the atomicity scope, we can have i) strong atomicity, which requires that transactions be executed in mutual exclusion with the rest of the program code (e.g., [17], [8], UTM [1], VTM [23]) or ii) weak atomicity which requires mutual exclusion only to other transactions, allowing the interleaved execution with the non-transactional code (e.g., [14], [27], LTM [1]).

III. GARBAGE COLLECTION STRATEGIES

The typical GC algorithm *called mark-and-sweep* [34] has two phases. First, live objects are distinguished from the garbage using tracing. Reachable objects are marked; either by altering bits within the objects, or by recording a bitmap. In the second phase, garbage is collected by examining all the unmarked objects and reclaiming their space; all the memory is then swept. The GC must run to completion without interruption because the system is in an unstable state until the garbage collection completes, which is rather “incompatible” with real-time (e.g., in current Java virtual machines, a thread may be blocked while the garbage collector is executing).

The fragmentation problem arises when there are objects of different sizes. Mark-and-sweep collectors solve the fragmentation problem by *compaction*. Marked objects are moved at one end of the memory heap, without disturbing the objects’ original order. Others variations of this algorithm avoids fragmentation by dealing with multiple areas for different size objects (i.e., *segregation*) [19].

1) Copying Collectors: Unlike mark-and-sweep collectors, a copying collector does not collect garbage, it copies all the live objects into one area, and the rest of the heap is free. The marking phase is thus coupled with the compaction process. The work needed is proportional to the amount of live memory. A very common approach is the semi-space collector [34], which subdivides the heap into two contiguous semi-spaces, called from-space and to-space. When the running program demands an allocation that out of memory by the current semi-space (i.e., the from-space), the program is stopped and all the live objects are copied from from-space to to-space. Then, the roles of the two semi-spaces are reversed. In this context, increasing the memory heap decreases the GC frequency, while it increases the average age of objects and the GC efficiency.

2) Incremental Copying Schemes: An incremental adaptation of the copying GC is given in [2]; a GC execution begins with an atomic flip that copies directly reachable objects from root pointers (e.g., a global variables) into the memory semi-space to the to-space. Then, any from-space object that is being accessed by the application is first copied into to-space. This copying-on-demand is called read barrier, and is interleaved with the program execution. New objects are allocated into to-space. Finally, the objects in the from-space are invalidated, and the roll of both semi-spaces changed.

A variant of this approach avoids checking whether a node is in from-space or in to-space by referencing a node through an indirection. If the indirection points to itself, the node is in to-space. If not, the node is in from-space and the indirection points to the new version of the node (the node in to-space). A development of this algorithm uses write barriers, instead of read barriers [3]. Write barriers consist to evacuate the node upon modification, instead of upon access.

3) Incremental non-Copying Collectors: The tri-color marking algorithm was introduced by Dijkstra et al. [6] to prove the correctness of cooperating programs. The algorithm assigns a color (i.e., black, white, or grey) to each node in the object graph. Initially, the root set is colored grey and all the other nodes white. The collector proceeds by scanning grey nodes for pointers to white nodes. The white nodes that are found are turned grey, and the grey nodes scanned are turned black. The collection is completed when there are no more grey objects. Then, the reachable part of the graph has been discovered, all the white nodes can then be recycled, and all the black nodes become white.

While the collector determines the reachable nodes, the application can concurrently access any part of the graph, and allocate new nodes or modify existing nodes. To synchronize the application and the collector no black nodes have a white child. The application must preserve this invariant by changing the colors of the nodes affected, if necessary. Hence, each tracing step involves a grey node from which white children are colored grey.

4) Read/Write Barriers: The read/write barriers can be implemented through software instructions preceding the read/write of potential pointers from the heap. Alternatively, checking read barriers can be implemented by a specialized hardware (i.e., [22]) or by microcode routines. Another approach uses the virtual memory system to have read barrier checks implicitly performed by the MMU hardware.

5) Real-time Garbage Collection: Incremental algorithm requires that all root pointers are scanned in a single atomic operation. A real-time garbage collector must fulfill two conflicting basic properties: ensure that programs with bounded allocation rates do not run out of memory, and provide short blocking times. Then, in order to use garbage collection, real-time system programmers are required to derive strict upper bounds on memory usage and allocation rates, resulting in the need for the application developer to be aware of memory management to an extent not needed for normal Java.

When using incremental garbage collectors, two major sources of blocking are exhibited: root scanning and heap compaction. Finding root nodes of an object graph is an integral part of garbage collector functionality and cannot be circumvented. Heap compaction is necessary to avoid unbounded heap fragmentation, which in turn would lead to unacceptably high memory consumption or application failure.

IV. TM-based Real-time Garbage Collection

Taken as started the incremental tri-color algorithm [6], we change lock-based synchronization by transactions, making each tracing step an atomic action. Each transaction involves the exploration of a node grey node, in order to grey its white children and whether committing coloring it black. A transaction is either aborted when detecting a conflict, or committed in case of successful completion.

Since conflicts are handled with non-blocking synchronization, this mechanism offers a stronger forward progress guarantees.

A. The TM Size and the Object Structure

The Sun JDK and SDK use a mark-and-sweep GC. In order to avoid fragmentation problems, the strategy of these JVMs relies on occasionally run a compacting GC. To reduce the cost of object relocation, each object has a non-moving handle that points to the location of the object header. When an object is relocated, its handle is updated. In this way, relocating objects is transparent to the application program, which always accesses objects using their nonmoving handle. The heap is organized in two sections: the handle space and the object space. Since the handle space consists of fixed size objects (i.e., two words of 32 bits, the first word provides a pointer to the object and the second, a pointer to the object's class), it does not need be compacted. If the piece is not free, the data of an object follows the header word. Compaction is made in two phases: the object space is first compacted, and the handles are then updated.

The Java Hotspot GC [33], provides an accurate GC, which eliminates object handles. Then, the collector must find and update all references to an object when the object is relocated. Indirect handles make relocating objects easier, but it introduces performance degradation because references to instance variables require two memory accesses. Eliminating handles improves also memory consumption in a word per object (approximately 8% of the total Java heap space). Every object occupies a number of bytes that is a multiple of 8. Then, whether the number of bytes required by an object (i.e., for its header and fields) is not a multiple of 8, it is round up to the next multiple of 8. For example, an instance of a class with a single `boolean` field takes up 16 bytes: 8 bytes of header, 1 byte for the `boolean` field and 7 bytes added to make the size up to a multiple of 8; while an instance with eight `boolean` fields, also takes 16 bytes.

In order to provide real-time guaranties for the GC, the JamaicaVM [31] divides the application objects in blocks of the same size (i.e., 32 bytes), that are explored by the GC in a bounded time. This strategy avoids also the fragmentation problem, because the heap is divided also in blocks of the same size, each one can allocate an object block.

Since the GC runs as lower priority than the application, the worst-case for conflict resolution is when the application must wait for the abortion of the operation making by the GC. In order to restrict both the worst-case for conflict detection and the worst-case for conflict resolution, we can limit the transaction size. The Jamaica VM from AICAS [31] offers us the scheme ideal to establish these limits without limiting the object size.

B. Using TM to Synchronize the GC

A comparison of wait-free and lock-free algorithms with spin-based and suspension-based synchronization in a Real-time environment is presented in [4]. As an interesting conclusion of this study, is that non-blocking algorithms perform better for small, simple shared objects.

Taken into account the small average size of Java objects that the specJVM applications present (i.e., about 32 Bytes) [11], we found interesting to use a Hardware-based TM solution to synchronize an incremental GC and the application. Since we can bound the WCET (e.g., by knowing the time of each committed transactions and the number of retries for the aborted transactions), we can establish a bounded time for each increment; having a real-time GC.

Depending on the granularity at which data is modified, TM systems can be either, word-based or object-based. Since our focus is to study a TM supporting the concurrent/parallel execution of a RT collector and a RT Java application, we deal with Object-based TM.

The TM is an optimistic synchronization approach, which increases the concurrency of the GC and the application, because the application threads do not needed to wait for accessing a node in the graph (i.e., an object), and both the GC and the application threads can simultaneously modify disjoint nodes of the graph structure in a safety way. By using look-based synchronization, mutual exclusion among the GC and the application is achieved by isolating the object accessed under the same lock (i.e., by using coarse-grain looking).

Note that when synchronizing the GC and the application by a TM technique, conflicts are very improbably because only arise when the GC scans the same object that application is modifying. Since conflicts among the GC and the application threads arise rarely enough, this technique allows us to obtain a high performance gain over lock-based protocols, in particular on large numbers of processors; despite the overhead of retrying transactions that fail.

Real-time collectors need to partition their root sets in order to both: avoid violations and keep their own transactions small, avoiding lost too much work when violations do occur.

C. The application and the GC. Write Barriers

While lock-based method systematically block all accesses to shared resources (i.e., to all objects in a Java concurrent/parallel GC), transactional memory allows several transaction to access resources in parallel. This is the key idea for our proposal. While the GC explores an object, the application can access any object without blocking.

The interaction of the GC and the application is limited and well known. The GC does not modify the object fields (i.e., the object graph of the application); the GC only affects the color of the object. Note that the object´s color is not part of the object information; it is an added field in the object header allowing the GC to differentiate reachable objects (i.e., alive objects) from non-reachable objects (i.e., garbage).

Since the collector does not modify the application objects, the interaction among the GC and the application is limited to the object color modifications. Regarding the application synchronization requirements, the object´s color is only modified by the write barrier code executed for each assignment statement (i.e., whether the application creates a pointer from a black object to a white one).

This case does not create transactional conflicts, because the color of the object which the GC is exploring is always grey[2]. Note that the object that the GC is exploring is always grey. Regarding the GC synchronization requirements, the object color is only modified whiting an atomic or transaction, coloring grey the white children and then coloring black the explored object.

Transactional conflicts are due to modifications in the object graph made by the application while the GC is tracing it. Then, only whether the application accesses the same object that the GC is exploring a transactional conflict can arise. Moreover, only whether the application makes a reference from a field within the object which the GC is exploring. That means that the possible conflict in the transaction is when the application executes an assignment sentence X.field=Y while the GC is exploring the X object.

D. Avoiding the Priority Inversion Problem

The *Real-Time Specification of Java* (RTSJ) [4] RTSJ [3] defines memory allocation and reclamation specifications that enable the use of real-time compliant garbage collection algorithms without prescribing any specific solution to the technique employed by the GC within the heap, and makes distinction between three main kinds of tasks:

1. *Low-priority tasks:* are tolerant with the GC.
2. *High-priority tasks:* cannot tolerate unbounded preemption latencies.
3. *Critical tasks:* cannot tolerate preemption latencies.

Whereas high-priority tasks require a real-time GC, critical tasks must not be affected by the GC, and as a consequence cannot access any object within the heap. The GC interleaves its execution with non-real-time threads and real-time threads, having GC a priority higher than non-real-time threads and lower than real-time threads. This priority scheme results in the ideal scenario where the *priority inversion problem* can arise.

The priority inversion is a phenomenon which occurs where a high-priority thread is forced to wait for the exclusive access to a shared resource that a low-priority thread is holding and a third task with mid-term priority ejects the task with low-priority. As consequence, the higher-priority task is forced to wait not only by the lower priority blocking the shared resource, but it also must wait for a third task with mid-term priority.

For a given transaction, the worst case behavior for both, CPU time and memory consumption, has been theoretically estimated as *O(n)* for n concurrent transactions running in the same time. By prioritizing TM, the worst case execution time is determined as follows: *O(1)* for the highest priority task (i.e., the transaction of this task always commit), *O(2)* for the next priority level task (i.e., the transaction of this task can be aborted only by the high priority task), and so on *(i.e., O(n)* for the lower priority task). However, the real value of this estimation depends on implementation details (e.g., we can avoid overhead by aborting early enough a transaction that fails).

E. Design Decision in a TM RT-Collector

In a real-time environment, the GC must run as lower priority than the real-time application threads. Then, in a conflictive case, the GC always aborts its transaction, while the application always continues its execution (i.e., commits). Taken into account the low probability of transactional conflicts when synchronizing the GC and the application, we consider the use of an eager police for data consistency (i.e., the *in-situ* value is the new one).

For conflict detection, we must take into account only objects colored grey (i.e., there are not conflicts with black or white objects). The GC performs an incremental step as an atomic operation (i.e., a transaction).

Each GC step consists to explore a grey object (e.g., X) coloring its children grey and blackening the object. A transactional conflict arises whether the application makes an assignment from X to a white object. Note that references to grey or black objects do not affects the semantics of the GC and, as consequence, does not creates transactional conflicts.

Moreover, whether the referenced object is white and it is assigned to a field still not explored there is not transactional conflict, because this case does not affect the reachable propriety of the object within the graph explored by the GC. Note that as part of the transaction, the referenced object (i.e., the Y object) will colored grey when arriving to the corresponding field (i.e., X.field). Then, there is transactional conflict only whether assigning a white object into a field which has been explored yet. In this case, the GC transaction must be aborted, while the application continues its execution. Note that we can avoid this

[2] The color black means that the object has been yet explored, while the color white means that the object is still not reached.

conflictive case by modifying the write barrier code in order to avoid the application to create a pointer from a grey object to a white one. Then, the referenced object is colored grey. Then, we modify the write barrier code executed for each assignment statement as shown Figure 1.

```
If ((color(X)<>white)&&(color(y)==white)
          Y.color = grey;
```

Figure 1. Write barrier code for X=Y or X.field=Y.

Note that this solution does not modify the tricolor invariant: "*there are not pointers from black objects to white ones*"; because it can be references from grey objects to white ones. This modification has been introduced in [31] in order to obtain a parallel collector with real-time guaranties.

Taken into account the adaptive GC presented in [36], which changes its priority as the free memory amount needed changes, we must to adapt the TM conflict resolution politic. When the free memory in the heap drops below a determined parameter, RTGC is invoked at the normal priority (i.e., non-real-time threads can be preempted by the RTGC). If it falls below a dangerous limit the RTGC priority increases (i.e., soft-real-time threads can be preempted by the RTGC). Further, if the available memory falls below a critical limit, all non-critical-real-time threads (i.e., soft-real-time and non-real-time threads) are blocked to protect critical tasks.

Then, where the GC can boost its priority making it non-preemptible by real-time threads (only critical real-time threads can preempt the GC with boosted priority), we must to adapt the conflict resolution politic of our proposed solution in order to abort the application transaction instead of the collector which commits.

V. Conclusions

The TM synchronized models simplify writing parallel and concurrent applications. It simplifies concurrent programs allowing higher concurrency than lock-based synchronization. While standard implementations of TM are optimized for the average case, for real-time system we require to optimize the worst case execution time. The hardware-based implementation of TM is new paradigm of synchronization for multiprocessors.

Since the GC and the application can be modeled as two different threads with a large shared data space, but minimal exclusion and synchronization constraints; transactional memory results a more optimistic approach than look-based techniques, allowing the application and the GC to simultaneously modify in a safety way different objects typically protected under the same lock.

For real-time systems, transactional memory techniques avoid the problem of the priority inversion naturally, because just this does not occur. We believe that real-time systems can benefit greatly of the advantages of TM, and that TM and GC present an interesting analogy. Thus, we are studying the possibility to use HTM to synchronize the execution of parallel collectors in embedded Java environments. We also believe that by using a simple specialized hardware support and to adapting the GC algorithm, we can offer predictive execution and time guarantees for real-time systems.

TM-based collection is an area of future interest for us. Since Java is memory safe, that means it prevents memory accesses through un-typed pointers, allowing the implementation of copying-based GC. As an alternative to the Jamaica real-time tricolour-based algorithm, we can investigate a real-time copying-based GC using HTM synchronization on a multi-core system.

Acknowledgment

This research was supported by Consejería de Educación de Comunidad de Madrid, through Research Program S2009/TIC - 1468, and by Ministerio de Educación y Ciencia, through the research grant TIN2009 - /07146.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. "Unbounded Transactional memory". In Proc. of the the 11th International Symposium on High-Performance Computer Architecture. HPCA 2005.

[2] H. Baker. "List Processing in Real Time on a Serial Computer". Communications of the ACM, 21(4):280–294, April 1978.

[3] H. Baker. "The Treadmill: Real-Time Garbage Collection without Motion Sickness". In Proc. of the Workshop on Garbage Collection in Object-Oriented Systems. OOPSLA'91, 1991. Also appears as SIGPLAN Notices 27(3), pages 66-70, March 1992.

[4] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time Synchronization on Multiprocessors: To block or not to block, to suspend or spin?" in IEEE Real-Time and Embedded Technology and Applications Symposium. RTAS 2008.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. "Bulk Disambiguation of Speculative Threads in Multiprocessors". In Proc. of the of 33th International Symposium on Computer Architecture. ISCA 2006.

[6] E. Dijkstra, L. Lamport, A. Martin, and C. S. E. Steffens. "On-the-fly Garbage Collection: An Exercise in Cooperation". Communications of the ACM, 21(11): 965–975, November 1978.

[7] D. Grossman. "The Transactional Memory / Garbage Collection Analogy". In Proc. of Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2007

[8] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, Ch. Kozyrakis, and K. Olukotun. "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software". IEEE Micro, 24(6), 2004.

[9] T. Harris, J. Larus, and R. Rajwar. “Transactional Memory “. 2on edition. Morgan &Claypool Plubishers.

[10] Haskell Communities.”The Glasgow Haskell Compiler”. http://www.haskell.org/ghc/

[11] M. Teresa Higuera-Toledano and Valérie Issarny. “Analyzing the Performance of Memory Management in RTSJ”. In Proc. of the 5th International on Object-oriented Real-time distributed Computing. ISORC 2002.

[12] M. Hirzel and P. Nagpurkar. “Dualities in Programming Languages”. In Proc. of ACM Conference on Programming Language Design and Implementation, PLDI 2010.

[13] Ch. C. Minh, M. Trautmann, J. W. Chung, A. McDonald, N. Bronson, J. Casper, Ch. Kozyrakis, and K. Olukotun. “An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees”. International Symposium on Computer Architecture. ISCA 2007.

[14] M. Herlihy and E. B. Moss. “Transactional Memory: Architectural Support for Lock-free data Structures”. In Proc. of the of 20nd International Symposium on Computer Architecture. ISCA 1993.

[15] K. Nikas, N. Anastopoulos and, G. Goumas N. Koziris. ”Employing Transactional Memory and Helper Threads to Speedup Dijkstra’s Algorithm.” In Proc. 38th International Conference on Parallel Processing, 2009.

[16] N. Anastopoulos, G. Goumas, K. Nikas and N. Koziris. “Early Experiences on Accelerating Dijkstra’s Algorithm Using Transactional Memory”. In Proc. of 23rd IEEE International Symposium on Parallel and Distributed Processing. IPDPS 2009.

[17] T. F. Knight. “An Architecture for Mostly Functional Languages”. In Proc. of ACM Lisp and Functional Programming Conference 1986.

[18] E.K. Kolodner and W.E Weihl. “Atomic Incremental Garbage Collection and Recovery for Large Stable Heap.”. In Proc. ACM International Conference on the Management of Data. SIGMOD 1993.

[19] T. Lim, P. Pardyak, and B. Bershad.”A Memory-Efficient Real-Time non-Copying Garbage Collector”. In Proc. of ACM International Symposium on Memory Management,pages 118–129. ACM Digital Library, 1998.

[20] D.B. Lomet. ”Process Structuring, Synchronization, and Recovery using Atomic Actions”. In Proc. of Language Design for Reliable Software “, 1977.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D.A. Wood. “Logtm: Log-based Transactional Memory”. International Symposium on High-Performance Computer Architecture. HPCA 2006.

[22] Sun Microsystems. picoJava-II Programmer’s Reference Manual. Technical report, http://www.sun.com/microelectronics/picoJava 1999.

[23] R. Rajwar, M. Herlihy, and K. Lai. “Virtualizing Transactional Memory”. In Proc. of the of 32nd International Symposium on Computer Architecture. ISCA 2005.

[24] T. Riegel, C. Fetzer and P. Felber. ”Time-based Transactional Memory with Scalable Time Bases”. In Proc. of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA 2007.

[25] M. Schoeberl, F. Brandner and J. Vitek. ”RTTM: Real-time Transactional Memory”. In Proc. of SAC 2010

[26] The Real-Time for Java Expert Group. “Real-Time Specification for Java” . ADDISON-WESLEY, 2000.

[27] N. Shavit and D. Touitou. “Software Transactional Memory”. In Proc. of the 14th ACM Symposium on Principles of Distributed Computing 1995.

[28] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. “An Integrated Hardware-Software Approach to Flexible Transactional Memory”. International Symposium on Computer Architecture. ISCA 2007.