C7: FAST SOFTWARE TRANSACTIONAL MEMORY FOR DYNAMIC LANGUAGES

Remigius Meier

Department of Computer Science
ETH Zürich, Switzerland
remi.meier@inf.ethz.ch

Armin Rigo

www.pypy.org
arigo@tunes.org

ABSTRACT

...

1. INTRODUCTION

Dynamic languages like Python, PHP, Ruby, and JavaScript are usually regarded as very expressive but also very slow. In recent years, the introduction of just-in-time compilers (JIT) for these languages (e.g. PyPy, V8, Tracemonkey) started to change this perception by delivering good performance that enables new applications. However, a parallel programming model was not part of the design of those languages. Thus, the reference implementations of e.g. Python and Ruby use a single, global interpreter lock (GIL) to serialize the execution of code in threads.

While this GIL prevents any parallelism from occurring, it also provides some useful guarantees. Since this lock is always acquired while executing bytecode instructions and it may only be released in-between such instructions, it provides perfect isolation and atomicity between multiple threads for a series of instructions. Another technology that can provide the same guarantees is transactional memory (TM).

There have been several attempts at replacing the GIL with TM. Using transactions to enclose multiple bytecode instructions, we can get the very same semantics as the GIL while possibly executing several transactions in parallel. Furthermore, by exposing these interpreter-level transactions to the application in the form of *atomic blocks*, we give dynamic languages a new synchronization mechanism that avoids several of the problems of locks as they are used now.

Our contributions include:

- We introduce a new software transactional memory (STM) system that performs well even on low numbers of CPUs. It uses a novel combination of hardware features and garbage collector (GC) integration in order to keep the overhead of STM very low.
- This new STM system is used to replace the GIL in Python and is then evaluated extensively.
- We introduce atomic blocks to the Python language to provide a backwards compatible, composable synchronization mechanism for threads.

2. BACKGROUND

2.1. Transactional Memory

Transactional memory (TM) is a concurrency control mechanism that comes from database systems. Using transactions, we can group a series of instructions performing operations on memory and make them happen atomically and in complete isolations from other transactions. *Atomicity* means that all these instructions in the transaction and their effects seem to happen at one, undividable point in time. Other transactions never see inconsistent state of a partially executed transaction which is called *isolation*.

If we start multiple such transactions in multiple threads, the TM system guarantees that the outcome of running the transactions is *serializable*. Meaning, the outcome is equal to some sequential execution of these transactions. Overall, this is exactly what a single global lock guarantees while still allowing the TM system to run transactions in parallel as an optimization.

2.2. Python

We implement and evaluate our system for the Python language. For the actual implementation, we chose the PyPy interpreter because replacing the GIL there with a TM system is just a matter of adding a new transformation to the translation process of the interpreter.

Over the years, Python added multiple ways to provide concurrency and parallelism to its applications. We want to highlight two of them, namely *threading* and *multiprocessing*.

Threading employs operating system (OS) threads to provide concurrency. It is, however, limited by the GIL and thus does not provide parallelism. At this point we should mention that it is indeed possible to run external functions written in C instead of Python in parallel. Our work focuses on Python itself and ignores this aspect as it requires writing in a different language.

The second approach, *multiprocessing*, uses multiple instances of the interpreter itself and runs them in separate OS processes. Here we actually get parallelism because we have one GIL per interpreter, but of course we have the overhead of multiple processes / interpreters and also need to exchange data between them explicitly and expensively.

We focus on the *threading* approach. This requires us to remove the GIL from our interpreter in order to run code in parallel on multiple threads. One approach to this is fine-grained locking instead of a single global lock. Jython and IronPython are implementations of this. It requires great care in order to avoid deadlocks, which is why we follow the TM approach that provides a *direct* replacement for the GIL. It does not require careful placing of locks in the right spots. We will compare our work with Jython for evaluation.

2.3. Synchronization

It is well known that using locks to synchronize multiple threads is hard. They are non-composable, have overhead, may deadlock, limit scalability, and overall add a lot of complexity. For a better parallel programming model for dynamic languages, we want to add another, well-known synchronization mechanism: *atomic blocks*.

Atomic blocks are composable, deadlock-free, higher-level and expose useful atomicity and isolation guarantees to the application for a series of instructions. An implementation using a GIL would simply guarantee that the GIL is not released during the execution of the atomic block. Using TM, we have the same effect by guaranteeing that all instructions in an atomic block are executed inside a single transaction.

Remi ▶*STM, how atomicity & isolation; reasons for overhead*◀

3. METHOD

3.1. Transactional Memory Model

In this section, we describe the general model of our TM system. This should clarify the general semantics using commonly used terms from the literature.

3.1.1. Conflict Handling

Our conflict detection works with *object granularity*. Conceptually, it is based on *read* and *write sets* of transactions. Two transactions conflict if they have accessed a common object that is now in the write set of at least one of them.

The *concurrency control* works partly *optimistically* for reading of objects, where conflicts caused by just reading an object in transactions are detected only when the transaction that writes the object actually commits. For write-write conflicts we are currently *pessimistic*: Only one transaction may have a certain object in its write set at any point in time, others trying to write to it will have to wait or abort.

We use *lazy version management* to ensure that modifications by a transaction are not visible to another transaction before the former commits.

3.1.2. Semantics

As required for TM systems, we guarantee complete *isolation* and *atomicity* for transactions at all times. Furthermore, the isolation provides full *opacity* to always guarantee a consistent read set.

We support the notion of *inevitable transactions* that are always guaranteed to commit. There is always at most one such transaction running in the system. We use this kind of transaction to provide *strong isolation* by running non-transactional code in the context of inevitable transactions and to still provide the *serializability* of all transaction schedules.

3.1.3. Contention Management

When a conflict is detected, we perform some simple contention management. First, inevitable transactions always win. Second, the older transaction wins. Different schemes are possible.

3.1.4. Software Transactional Memory

Generally speaking, the system is fully implemented in software. However, we exploit some more advanced features of current CPUs, especially *memory segmentation, virtual memory,* and the 64-bit address space.

3.2. Implementation

In this section, we will present the general idea of how the TM model is implemented. Especially the aspects of providing isolation and atomicity, as well as conflict detection are explained. We try to do this without going into too much detail about the implementation. The later section 3.3 will discuss it in more depth.

3.2.1. Memory Segmentation

A naive approach to providing complete isolation between threads is to partition the virtual memory of a process into N segments, one per thread. Each segment then holds a copy of all the memory available to the program. Thus, each thread automatically has a private copy of every object that it can modify in complete isolation from other threads.

To get references to objects that are valid in all threads, we will use the object's offset inside the segment. Since all segments are copies of each other, the *Segment Offset (SO)* will point to the private version of an object in all threads/segments. To then translate this SO to a real virtual memory address when used inside a thread, we need to add the thread's segment start address to the SO. The result of this operation is called a *Linear Address (LA)*. This is illustrated in Figure 1.

To make this address translation efficient, we use the segment register $\%gs$. When this register points to a thread's segment start address, we can instruct the CPU to perform the above translation from a reference of the form $\%gs{::}SO$ to the right LA on its own.

In summary, we can use a single SO to reference the same object in all threads, and it will be translated by the CPU to a LA that always points to the thread's private version of this object. Thereby, threads are fully isolated from each other. However, N segments require N-times the memory and modifications on an object need to be propagated to all segments.

3.2.2. Page Sharing

In order to eliminate the prohibitive memory requirements of keeping around N segment copies, we share memory between them. The segments are initially allocated in a single range of virtual memory by a call to `mmap()`. As illustrated in Figure 2, `mmap()` creates a mapping between a range of virtual memory pages and virtual file pages. The virtual file pages are then mapped lazily by the kernel to real physical memory pages. The mapping generated by `mmap()` is initially linear but can be changed arbitrarily. Especially, we can remap so that multiple virtual memory pages map to a single virtual file page. This is what we use to share memory between the segments since then we also only require one page of physical memory.

As illustrated in Figure 3, in our initial configuration (I) all segments are backed by their own range of virtual file pages. This is the share-nothing configuration.

We then designate segment 0 to be the *Sharing-Segment*. No thread gets this segment assigned to it, it simply holds the pages shared between all threads. So in (II), we remap all virtual pages

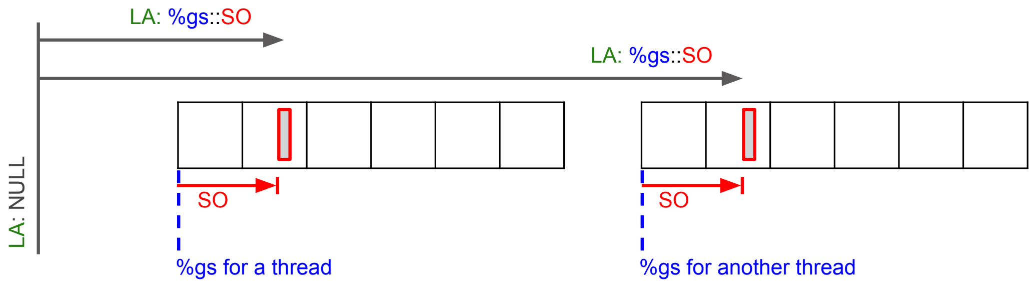

Fig. 1. Segment Addressing

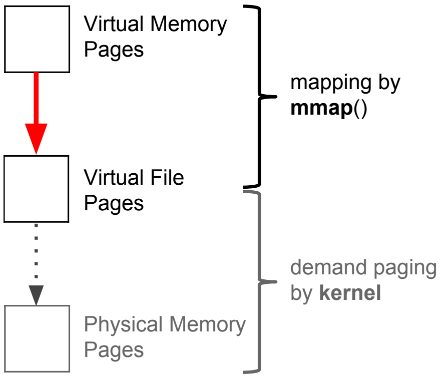

Fig. 2. `mmap()` Page Mapping

of the segments > 0 to the file pages of our sharing-segment. This is the fully-shared configuration.

During runtime, we can then privatize single pages in segments > 0 again by remapping single pages as seen in (III).

Looking back at address translation for object references, we see now that this is actually a two-step process. First, $\%gs{::}SO$ gets translated to different linear addresses in different threads by the CPU. Then, depending on the current mapping of virtual pages to file pages, these LAs can map to a single file page in the sharing-segment, or to privatized file pages in the corresponding segments. This mapping is also performed efficiently by the CPU and can easily be done on every access to an object.

In summary, $\%gs{::}SO$ is translated efficiently by the CPU to either a physical memory location which is shared between several threads/segments, or to a location in memory private to the segment/thread. This makes the memory segmentation model for isolation memory efficient again.

3.2.3. Isolation: Copy-On-Write

We now use these mechanisms to provide isolation for transactions. Using write barriers, we implement a *Copy-On-Write (COW)* on the level of pages. Starting from the initial fully-shared configuration (Figure 3, (II)), when we need to modify an object without other threads seeing the changes immediately, we ensure that all pages belonging to the object are private to our segment.

To detect when to privatize pages, we use write barriers before every write. When the barrier detects that the object is not in a private page (or any pages that belong to the object), we remap and copy the pages to the thread's segment. From now on, the translation of $\%gs{::}SO$ in this particular segment will resolve to the private version of the object. Note, the SO used to reference the object does not change during that process.

3.2.4. Isolation: Barriers

The job of barriers is to ensure complete isolation between transactions and to register the objects in the read or write set. We insert read and write barriers before reading or modifying an object except if we statically know an object to be readable or writable already.

Read Barrier: Adds the object to the read set of the current transaction. Since our two-step address translation automatically resolves the reference to the private version of the object on every access anyway, this is not the job of the read barrier anymore.

Write Barrier: Adds the object to the read and write set of the current transaction and checks if all pages of the object are private, doing COW otherwise.
Furthermore, we currently allow only one transaction modifying an object at a time. To ensure this, we acquire a write lock on the object and also eagerly check for a write-write conflict at this point. If there is a conflict, we do some contention management to decide which transaction has to wait or abort. Eagerly detecting this kind of conflict is not inherent to our system, future experiments may show that we want to lift this restriction.

3.2.5. Atomicity: Commit & Abort

To provide atomicity for a transaction, we want to make changes visible on commit. We also need to be able to completely abort a transaction without a trace, like it never happened.

Commit: If a transaction commits, we synchronize all threads so that all of them are waiting in a safe point. In the committing transaction, we go through all objects in the write set and check if another transaction in a different segment read the same object. Conflicts are resolved again by either the committing or the other transaction waiting or aborting.
We then push all changes of modified objects in private pages to all the pages in other segments, including the sharing-segment (segment 0).

Abort: On abort the transaction will forget about all the changes it has done. All objects in the write set are reset by copying their previous version from the sharing-segment into the private pages of the aborting transaction.

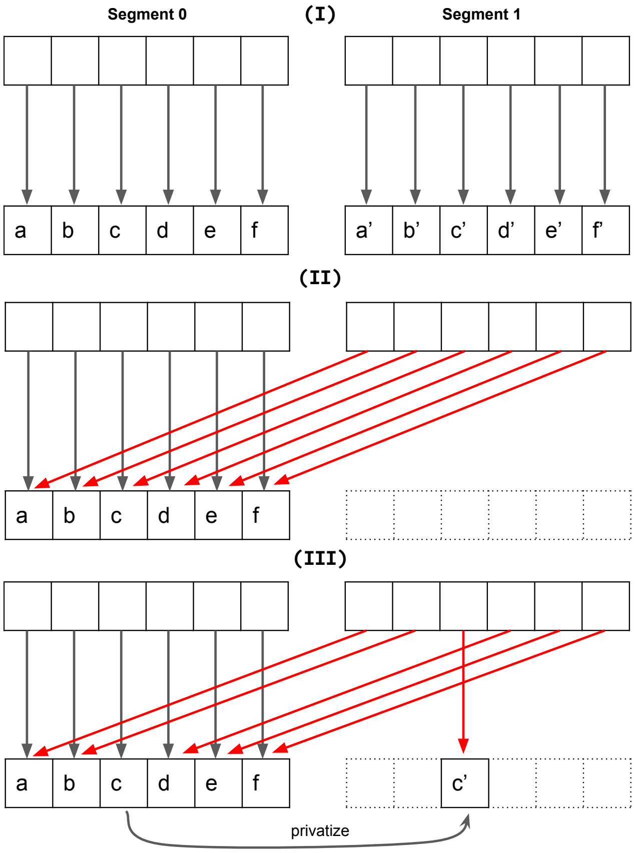

Fig. 3. Page Remapping: (I) after `mmap()`. (II) remap all pages to segment 0, fully shared memory configuration. (III) privatize single pages.

3.2.6. Summary

We provide isolation between transactions by privatizing the pages of the segments belonging to the threads the transactions run in. To detect when and which pages need privatization, we use write barriers that trigger a COW of one or several pages. Conflicts, however, are detected on the level of objects; based on the concept of read and write sets. Barriers before reading and writing add objects to the corresponding set; particularly detecting write-write conflicts eagerly. On commit, we resolve read-write conflicts and push modifications to other segments. Aborting transactions simply undo their changes by copying from the sharing-segment.

3.3. Low-level Implementation

In this section, we will provide details about the actual implementation of the system and discuss some of the issues that we encountered.

3.3.1. Architecture

Our TM system is designed as a library that covers all aspects around transactions and object management. The library consists of two parts: (I) It provides a simple interface to starting and committing transactions, as well as the required read and write barriers. (II) It also includes a *garbage collector (GC)* that is closely integrated with the TM part (e.g. it shares the write barrier). The close integration helps in order to know more about the lifetime of an object, as will be explained in the following sections.

3.3.2. Application Programming Interface

```
void stm_start_transaction(tl, jmpbuf)
void stm_commit_transaction()
void stm_read(object_t *obj)
void stm_write(object_t *obj)
object_t *stm_allocate(ssize_t size_rounded)
STM_PUSH_ROOT(tl, obj)
STM_POP_ROOT(tl, obj)
```

`stm_start_transaction()` starts a transaction. It requires two arguments, the first being a thread-local data structure and the second a buffer for use by `setjmp()`. `stm_commit_transaction()` tries to commit the current transaction. `stm_read()`, `stm_write()` perform a read or a write barrier on an object and `stm_allocate()` allocates a new object with the specified size (must be a multiple of 16). `STM_PUSH_ROOT()` and `STM_POP_ROOT()` push and pop objects on the shadow stack [1]. Objects have to be saved using this stack around calls that may cause a GC cycle to happen, and also while there is no transaction running. In this simplified API, only `stm_allocate()` and `stm_commit_transaction()` require saving object references.

The type `object_t` is special as it causes the compiler [2] to make all accesses through it relative to the $\%gs$ register. With exceptions, nearly all accesses to objects managed by the TM system should use this type so that the CPU will translate the reference to the right version of the object.

3.3.3. Setup

On startup, we reserve a big range of virtual memory with a call to `mmap()` and partition this space into $N + 1$ segments. We want to run N threads in parallel while segment 0 is designated as the *sharing-segment* that is never assigned to a thread.

The next step involves using `remap_file_pages()`, a Linux system call, to establish the fully-shared configuration. All pages of segments > 0 map to the pages of the sharing-segment.

However, the layout of a segment is not uniform and we actually privatize a few areas again right away. These areas are illustrated in Figure 4 and explained here:

NULL page: This page is unmapped and will produce a segmentation violation when accessed. We use this to detect erroneous dereferencing of `NULL` references. All $\%gs::SO$ translated to linear addresses will point to NULL pages if SO is set to `NULL`.

Segment-local data: Some area private to the segment that contains segment-local information.

Read markers: These are pages that store information about which objects were read in the current transaction running in this segment.

Nursery: This area contains all the freshly allocated objects (*young objects*) of the current transaction. The GC uses pointer-bump allocation in this area to allocate objects in the first generation.

[1] A stack for pointers to GC objects that allows for precise garbage collection. All objects on that stack are never seen as garbage and are thus always kept alive.

[2] Clang 3.5 with some patches to this address-space 256 feature

Old object space: These pages are the ones that are really shared between segments. They mostly contain old objects but also some young ones that were too big to allocate in the nursery.

3.3.4. Assigning Segments

From the above setup it is clear that the number of segments is statically set to some N. That means that at any point in time, a maximum of N threads and their transactions can be running in parallel. To support an unlimited number of threads in applications that use this TM system, we assign segments dynamically to threads.

At the start of a transaction, the thread it is running in acquires a segment. It may have to wait until another thread finishes its transaction and releases a segment. Fairness is not guaranteed yet, as we simply assume a fair scheduling policy in the operating system when waiting on a condition variable.

Therefore, a thread may be assigned to different segments each time it starts a transaction. Although, we try to assign it the same segment again if possible. And a maximum of N transactions may run in parallel.

3.3.5. Garbage Collection

Garbage collection plays a big role in our TM system. The GC is generational and has two generations.

The **first generation**, where objects are considered to be *young* and reside in the *Nursery*, is collected by *minor collections*. These collections move the surviving objects out of the nursery into the old object space, which can be done without stopping other threads. This is done either if the nursery has no space left anymore or if we are committing the current transaction. Consequently, all objects are old and the nursery empty after a transaction commits. Furthermore, all objects in the nursery were always created in the current transaction. This fact is useful since we do not need to call any barrier on this kind of objects.

To improve this situation even more, we introduce the concept of *overflow objects*. If a minor collection needs to occur during a transaction, we empty the nursery and mark each surviving object in the old object space with an `overflow_number` globally unique to the current transaction. That way we can still detect in a medium-fast path inside barriers that the object still belongs to the current transaction.

The **second generation**, where objects are considered to be *old* and never move again, is collected by *major collections*. These collections are implemented in a stop-the-world kind of way and first force minor collections in all threads. The major goal is to free objects in the old objects space. Furthermore, we optimistically re-share pages that do not need to be private anymore.

As seen in the API (section 3.3.2), we use a *shadow stack* in order to provide precise garbage collection. Any time we call a function that possibly triggers a collection, we need to save the objects that we need afterwards on the shadow stack using `STM_PUSH_ROOT()`. That way, they will not be freed. And in case they were young, we get their new location in the old object space when getting them back from the stack using `STM_POP_ROOT()`.

3.3.6. Read Barrier

The point of the read barrier is to add the object to the read set of the transaction. This information is needed to detect conflicts between transactions. Usually, it also resolves an object reference to a private copy, but since the CPU performs our address translation on every object access efficiently, we do not need to do that in our barrier.

To add the object to the read set, for us it is enough to mark it as read. Since this information needs to be local to the segment, we need to store it in private pages. The area is called *read markers* and already mentioned in section 3.3.3. This area can be seen as a continuous array of bytes that is indexed from the start of the segment by an object's reference (SO) divided by 16 (this requires objects of at least 16 bytes in size). Instead of just setting the byte to `true` if the corresponding object was read, we set it to a `read_version` belonging to the transaction, which will be incremented on each commit. Thereby, we can avoid resetting the bytes to `false` on commit and only need to do this every 255 transactions. The whole code for the barrier is easily optimizable for compilers as well as perfectly predictable for CPUs:

```
void stm_read(SO):
    *(SO >> 4) = read_version
```

3.3.7. Write Barrier

The job of the write barrier is twofold: first, it serves as a write barrier for the garbage collector and second, it supports copy-on-write and adds objects to the write set of the transaction.

The **fast path** of the write barrier is very simple. We only need to check for the flag `WRITE_BARRIER` in the object's header and call the slow path if it is set. This flag is set either if the object is old and comes from an earlier transaction, or if there was a minor collection which will add the flag again on all objects. It is never set on freshly allocated objects that still reside in the nursery.

```
void stm_write(SO):
    if SO->flags & WRITE_BARRIER:
        write_slowpath(SO)
```

The **slow path** is shown here:

```
void write_slowpath(SO):
    // GC part:
    list_append(to_trace, SO)
    if is_overflow_obj(SO):
        SO->flags &= ~WRITE_BARRIER
        return
    // STM part
    stm_read(SO)
    lock_idx = SO >> 4
  retry:
    if write_locks[lock_idx] == our_num:
        // we already own it
    else if write_locks[lock_idx] == 0:
        if cmp_and_swap(&write_locks[lock_idx],
                        0, our_num):
            list_append(modified_old_objects, SO)
            privatize_pages(SO)
        else:
            goto retry
    else:
        w_w_contention_management()
        goto retry
    SO->flags &= ~WRITE_BARRIER
```

First comes the *GC part*: In any case, the object will be added to the list of objects that need tracing in the next minor collection (`to_trace`). This is necessary in case we write a reference to it that points to a young object. We then need to trace it during the next minor collection in order to mark the young object alive and

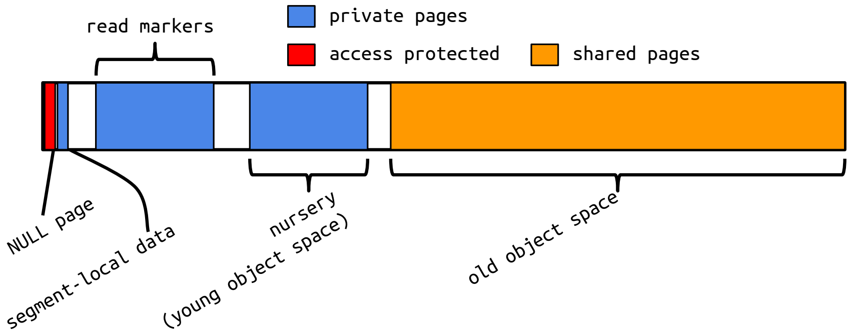

Fig. 4. Segment Layout

to update its reference to the new location it gets moved to. The check for `is_overflow_obj()` tells us if the object was actually created in this transaction. In that case, we do not need to execute the following *TM part*. We especially do not need to privatize the page since no other transaction knows about these "old" objects.

For TM, we first perform a read barrier on the object. We then try to acquire its write lock. `write_locks` again is a simple global array of bytes that is indexed with the SO of the object divided by 16. If we already own the lock, we are done. If someone else owns the lock, we will do a write-write contention management that will abort either us or the current owner of the object. If we succeed in acquiring the lock using an atomic `cmp_and_swap`, we need to add the object to the write set (a simple list called `modified_old_objects`) and privatize all pages belonging to it (copy-on-write).

In all cases, we remove the `WRITE_BARRIER` flag from the object before we return. Thus, we never trigger the slow path again before we do the next minor collection (also part of a commit) or we start the next transaction.

3.3.8. Abort

Aborting a transaction is rather easy. The first step is to reset the nursery and all associated data structures. The second step is to go over all objects in the write set (`modified_old_objects`) and reset any modifications in our private pages by copying from the sharing-segment. What is left is to use `longjmp()` to jump back to the location initialized by a `setjmp()` in `stm_start_transaction()`. Increasing the `read_version` is also done there.

3.3.9. Commit

Committing a transaction needs a bit more work. First, we synchronize all threads so that the committing one is the only one running and all the others are waiting in a safe point. We then go through the write set (`modified_old_objects`) and check the corresponding `read_markers` in other threads/segments. If we detect a read-write conflict, we do contention management to either abort us or the other transaction, or to simply wait a bit.

After verifying that there are no conflicts anymore, we copy all our changes done to the objects in the write set to all other segments, including the sharing-segment. This is safe since we synchronized all threads. We also need to push overflow objects generated by minor collections to other segments, since they may reside partially in private pages. At that point we also get a new `overflow_number` by increasing a global one, so that it stays globally unique for each transaction. Increasing the `read_version` is then done at the start of a new transaction.

3.3.10. Thread Synchronization

A requirement for performing a commit is to synchronize all threads so that we can safely update objects in other segments. To make this synchronization fast and cheap, we do not want to insert an additional check regularly in order to see if synchronization is requested. We use a trick relying on the fact that dynamic languages are usually very high-level and thus allocate a lot of objects very regularly. This is done through the function `stm_allocate` shown below:

```
object_t *stm_allocate(ssize_t size_rounded):
    result = nursery_current
    nursery_current += size_rounded
    if nursery_current > nursery_end:
        return allocate_slowpath(size_rounded)
    return result
```

This code does simple pointer-bump allocation in the nursery. If there is still space left in the nursery, we return `nursery_current` and bump it up by `size_rounded`. The interesting part is the check `nursery_current > nursery_end` which will trigger the slow path of the function to possibly perform a minor collection in order to free up space in the nursery.

If we want to synchronize all threads, we can rely on this check being performed regularly. So what we do is to set the `nursery_end` to 0 in all segments that we want to synchronize. The mentioned check will then fail in those segments and call the slow path. In `allocate_slowpath` they can simply check for this condition and enter a safe point.

For other synchronization requirements, for example:

- waiting for a segment to be released,
- waiting for a transaction to abort or commit,
- waiting for all threads to reach their safe points,

we use a set of condition variables to wait or signal other threads.

3.3.11. Contention Management

On encountering conflicts, we employ contention management to solve the problem as well as we can. The general rules are:

- prefer transactions that started earlier to younger transactions

- to support *inevitable* transactions, we always prefer them to others since they cannot abort

We can either simply abort a transaction to let the other one succeed, or we can also wait until the other transaction committed. The latter is an interesting option if we are trying to commit a write and another transaction already read the object. We can then signal the other transaction to commit as soon as possible and wait. After waiting, there is now no conflict between our write and the already committed read anymore.

4. EXPERIMENTAL RESULTS

compare some programs between

- pypy
- pypy-jit
- pypy-stm
- pypy-stm-jit
- cpython
- jython

5. RELATED WORK

6. CONCLUSIONS

7. FURTHER COMMENTS

Some citation: [1]

8. REFERENCES

[1] N.J. Higham, *Handbook of Writing for Mathematical Sciences*, SIAM, 1998.