A Transactional Memory System for Parallel Python

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

Armin Rigo
www.pypy.org
arigo@tunes.org

Thomas Gross
Department of Computer Science
ETH Zürich, Switzerland
thomas.gross@inf.ethz.ch

Abstract

Since some years, the popularity of dynamic languages is on the rise. A common trait of many of these language's implementations is the use of a single, global interpreter lock (GIL) to synchronise the interpreter in a multithreading scenario. Since this lock effectively serialises the execution, applications can not make use of the increasing parallelism in current hardware.

In this paper, we present a software transactional memory (STM) system specifically designed for the purpose of replacing the GIL in dynamic language interpreters while keeping the semantics. The key idea is a close integration of STM with the garbage collector to approach atomicity, synchronisation, and memory management in a unified way. Additionally, we implement atomic blocks as a scalable synchronisation mechanism by exposing part of this STM system to applications.

The described system uses a combination of features present in current CPUs, so that the parallelisation benefits outweigh the system's overhead already on 2 threads when comparing two Python interpreters – one with STM and one with a GIL.

Keywords transactional memory, global interpreter lock, dynamic language, python, virtual machine

1. Introduction

Dynamic languages like Python, PHP, Ruby, and JavaScript receive a lot of attention but only provide limited forms of parallelism. A parallel programming model was not part of the initial design of those languages. Thus, the reference implementations of, e.g., Python and Ruby use a single, global interpreter lock (GIL) to serialise the execution of code in threads. The use of a single, global lock causes several disadvantages, and as multi-core processors become the de facto standard in platforms ranging from smart phones to data centres, the issue of parallel execution must be addressed by these dynamic languages.

Unfortunately the GIL plays a central role in current implementations. Even recent efforts to address the performance problems of dynamic languages with just-in-time compilers (JIT) (e.g., PyPy [1], V8 [2], IonMonkey [3]) keep the GIL; Jython (an implementation of Python that is based on the JVM) is a notable exception.

DLS 2014, to be supplied.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The standard model employed in these systems is fairly straightforward: there can be multiple threads, but each acquires the GIL, then performs one (or several) instruction, and subsequently releases the lock. While this setup limits the benefits due to parallelism, the use of a GIL also provides some crucial guarantees. Since this lock is always acquired while executing bytecode instructions and may be released only in-between such instructions, the GIL ensures perfect isolation between multiple threads for a series of instructions – executing them atomically. In addition, the GIL provides the application with a sequential consistency model [28].

A global interpreter lock assures the atomic execution of individual instructions but offers the programmer no help in managing concurrent data structures. Many parallel programs must maintain invariants involving multiple objects (or multiple fields of a single object). So to enable Python programs for parallel execution, we need a way to identify which blocks of data must be executed atomically and hardware or software support to effect this atomic execution. This is traditionally done with locks.

Atomic blocks [31, 32] allow languages to indicate the units of computation that must be executed atomically to ensure correct concurrent execution. Atomic blocks map nicely to transactional memory, which in turn provides a convenient mechanism to replace the GIL. Using transactions to enclose multiple bytecode instructions, we can get the very same semantics as provided by the GIL while possibly executing several transactions in parallel. And by exposing these interpreter-level transactions to the application, we avoid the well-known problems caused by locks as applications use atomic blocks instead of explicit acquire/release operations on locks [29, 30, 33].

There have been several attempts at replacing the GIL with TM [20, 24, 25]. We focus here on software based transactional memory (STM) because that provides the unique opportunity to consider the issues of atomicity, synchronisation, and memory management (resp. garbage collection) together. STM systems impose no a-priori bound on the size of an atomic block and provide therefore a flexibility that is absent from current hardware-based TM systems [4]. The downside of STM systems is runtime overhead [22, 23], which can be substantial despite recent efforts to lower it [21, 27]. In this paper, we describe how we manage to lower the overhead of an STM system so that it can serve as a viable replacement for the GIL by leveraging the demands of a dynamic language like Python to integrate the STM system with the garbage collector.

This paper makes the following contributions:

- We introduce a new STM system that performs well for dynamic language interpreters.
- We integrate the system closely with a garbage collector (GC) to lower the overhead of STM.

- This new STM system is used to replace the GIL in one implementation of Python and is then evaluated.

The STM described here is efficient and has low overhead and is therefore beneficial also for environments with a small number of threads or cores.

1.1 Application View

We implement and evaluate our system for the Python language. Python is a dynamic programming language that was originally designed without concurrency in mind. Its reference implementation, CPython [5], uses a GIL. 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 real parallelism. The only code that is not necessarily restricted by the GIL in CPython is code written in e.g. C instead of Python. External functions can choose to run without the GIL and thus, in parallel. We ignore this aspect here as it requires writing in a different language than Python and is therefore not relevant to our work.

The GIL can also only ensure correctness of the interpreter itself: applications using threads are still required to coordinate concurrent accesses to data structures using conventional methods – locks being the most common way.

The second approach, *multiprocessing*, uses multiple instances of the interpreter itself and runs them in separate OS processes. Here we actually get parallelism because there is 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 the interpreter to run code in parallel on multiple threads. One approach to this is fine-grained locking instead of a single, global lock. Jython [6] and IronPython [7] follow this approach. Fine-grained locking is, however, not a *direct* replacement for the GIL. It requires multiple locks in multiple places, not only in a central location. We on the other hand follow the direct approach of using TM instead of the GIL. There, GIL acquisition and release operations conceptually directly map to starting and committing transactions.

TM also allows us to attack the issue of application-level synchronisation and atomicity. We introduce atomic blocks to the Python language by mapping them directly to the underlying transactions – providing the application with the guarantees of atomicity and isolation for the enclosed instructions.

2. Transactional Memory Model

In this section, we characterise the model of our TM system and its guarantees as well as some of the design choices we made. This should clarify the general semantics in commonly used terms from the literature [10].

The described TM system is fully implemented in software. However, we exploit some more advanced features of current CPUs, particularly *memory segmentation, virtual memory,* and the 64-bit address space. Still, it cannot be classified as a hybrid TM system since it currently makes no use of any hardware TM (HTM) present in the CPU.

2.1 Conflict Handling

We implement an object-based TM system, thus it makes sense to detect conflicts with *object granularity*. With this choice, if two transactions access the same object and at least one access is a write, we count it as a conflict. Conceptually, it is based on *read* and *write sets* of transactions. Reading from an object adds the object to the read set, writing to it adds it to both sets. Two transactions conflict if they have accessed a common object that is in the write set of at least one of them.

The detection, or *concurrency control*, works partly *optimistically* for reading objects. Read-write conflicts between two transactions are detected in both exactly at the time when the writing one 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. This decision needs to be evaluated further in the future.

When a conflict is detected, we perform some simple contention management that generally prefers the older transaction to the younger. This gives long transactions a better chance to succeed.

2.2 Semantics

As required for TM systems, we guarantee complete *isolation* and *atomicity* for transactions at all times. Our method of choice is *lazy version management*. Modifications by a transaction are not visible to another transaction before the former commits. Furthermore, the isolation provides full *opacity* [11] to always guarantee a consistent read set even for non-committed transactions.

To also support these properties for irreversible operations that cannot be undone when we abort a transaction (e.g. I/O, syscalls, and non-transactional code in general), we use *irrevocable* or *inevitable transactions* [12, 13]. These transactions are always guaranteed to commit, which is why they always have to win in case there is a conflict with another, normal transaction. There is always at most one such transaction running in the system, thus their execution is serialised. With this guarantee, providing *strong isolation* and *serialisability* between non-transactional code is possible by making the current transaction inevitable right before running irreversible operations.

3. High-Level Overview

In this section, we will present the general idea of how the TM model is implemented. The later section 4 will discuss it with more technical details.

3.1 Memory Segmentation and Page Sharing

Our STM system fundamentally builds on threads. Transactions run in their context, and there can be only one transaction per thread at a time. Transactions run in parallel because these threads can run in parallel. To isolate transactions from each other, we look at isolating the memory accesses from threads.

A naive approach to providing this complete isolation between threads is to partition the virtual memory of a process into N segments, one for each thread currently executing. Each segment then holds a complete copy of the memory logically seen by the program. All threads read and write data inside their own segment; and this data gets copied to other segments in an atomic fashion only when a transaction commits. So far, this is a model that would also work for distributed transactional memory.

Clearly, if we are in a shared-memory setting, this model is extremely wasteful: at any given time, most objects are identical in all, or most, segments. They differ only in segments that have modified but not yet committed these objects. So we make use of the Memory Management Unit (MMU) of the CPU. The MMU maps pages of virtual to physical memory. By changing this page mapping at the coarse unit of one page of memory (4096 bytes), we can share memory that would otherwise exist as identical copies in all segments.

In the common case, where objects are identical in all segments, we can thus make these pages *shared*. Conversely, when one

thread / segment wants to modify an object, all pages of this object need to be *privatised* for this segment. This involves duplicating the pages and changing the page mapping so that the private pages are only accessible by this segment and full isolation is guaranteed. This process is illustrated in Figure 1.

The result is that the virtual address space of the process is bloated by a factor N – but the real, physical memory usage is not, because most pages are shared with their identical copies in other segments.

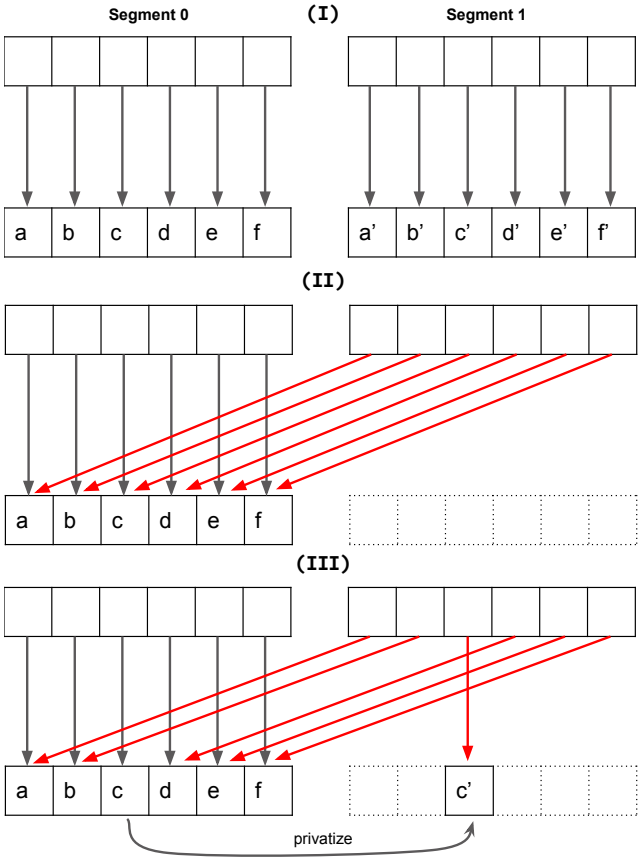

Figure 1. Page Remapping: (I) initial setting. (II) remap all pages to segment 0, fully shared memory configuration. (III) privatise single pages.

3.2 Memory Segmentation on x86 CPUs

When we store into our objects the references to other objects, we need to be careful: addresses that are valid in one segment are not valid in another, because they would break isolation by possibly pointing to the copy of the object in another segment. Consequently, we would need to adapt these addresses to the segment they live in, which would prevent completely any page sharing because then the binary contents differ between segments. Instead, we store pointers as *Segment Offsets (SO)*, i.e. offsets from the start of the segment. An SO can then be interpreted in any segment by adding to it the start address of that segment. The result is called a *Linear Address (LA)*. This is illustrated in Figure 2.

The x86 CPUs provide a feature which is called *memory segmentation*. On these CPU this is enough to perform the addition described above very efficiently. On the modern x86-64, there are two segment registers left: $\%fs$ and $\%gs$. On Linux for example, $\%fs$ is already used for addressing thread-local data efficiently. Thus, we use the remaining segment register $\%gs$, which is still available to applications, to point to the current thread's segment start address. The translation from an SO to an LA is done for us by using the "$\%gs:$ " prefix before any CPU instruction that reads or writes memory using an SO address. This is very efficient – we can do it on every object access – and some compilers support it natively (e.g. clang).

On non-x86 architectures, most simple memory accesses could still be done efficiently if the supported addressing modes allow for the addition of an offset stored in some register (e.g. ARM). For more complicated accesses (e.g. array indexing) or if the CPU does not support such an addressing mode, one extra addition may be required.

In summary, translating a $\%gs : SO$ to a physical address is a two-step process: First the memory segmentation feature of the CPU constructs a linear address. Then, this LA gets mapped by the MMU to some place in the physical memory. This makes the SO a valid reference to an object in all segments, automatically resolving either to a shared or a private version.

3.3 Conflict detection

What we described so far has no measurable performance overhead by itself, but it is missing conflict detection to be considered a full-fledged STM system. This requires adding *barriers* to the program to register the objects in the read or write set before reading or writing to them. We do this with an automatic, conservative, and local program transformation (cf. [26]). Furthermore, we still need to describe the commit protocol.

Read Barrier: A major contribution to our performance is the simplicity of the read barrier: a large program, like an interpreter for a complex language, contains a number of reads that typically vastly exceeds the number of writes. The described approach allows the read barrier to be a single, unconditional write into segment-local memory. We only need to *mark* an object as read by the current transaction. The mark is stored in a segment-local array indexed by a number derived from the SO of the object.

Unlike other STM systems, the read barrier does not have to find the private version of the object – our two-step address translation automatically resolves the reference to the private version on every access anyway.

Write Barrier: This barrier is triggered by checking a flag on the object that signifies if we did not already execute the write barrier on this object before. When the flag is set, we call the slow path, which adds the object into the write set and unconditionally resets the flag.

Here, we eagerly detect write-write conflicts by allowing only one transaction modifying an object at a time using write locks[1]. To make the object write-ready, this is also the place to privatise all pages the object belongs to, as described above.

Described below, we also need a write barrier for our generational garbage collector (GC). Interestingly, we are able to share the same barrier and flag for both use cases. Since the GC is an integral part of many dynamic languages, we often do not incur additional overhead since the barriers are already there. However, we do need some additional barriers when writing non-references into objects. In that case, generational GCs do not need a barrier but the STM system still does.

[1] Eager write-write conflict detection is not inherent to our approach; we may lift this restriction in the future if it proves to be useful.

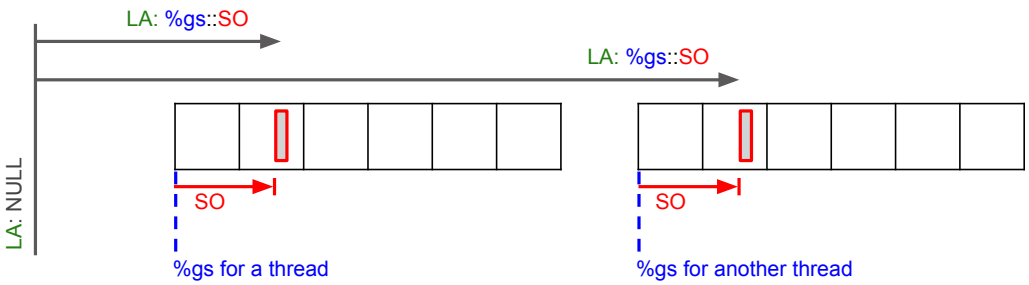

Figure 2. Segment Addressing

Commit: When we want to commit a transaction, we need to check for read-write conflicts. Such conflicts cannot be detected in the minimal read barrier, but can be detected during commit: we check if any of the objects in our write set was already read by another transaction. We never walk the read set, but just check if the objects are marked in other segments.

Checking other transactions' read marks would involve reading concurrently modified memory: another thread could be issuing a read barrier in parallel to a commit. To prevent this from occurring, we must suspend all other threads at known safe points. This is a trade-off for the simplicity of our read barrier – which is well worth it in our case, due to the very high number of read barriers when compared to the number of commits.

Conflicts can be resolved by aborting one of the two transactions, or in some cases by pausing the writing one until the reader commits – serialising the write after the read. If a commit succeeds, we copy the changes made to the objects in the write set so that they become visible in other segments. Note that this only includes objects that existed before the transaction and have been modified by it. It does not include all the new objects created by the current transaction, which typically outnumber modified objects by far in dynamic languages. We allocate these new objects in pages that are still shared, so that they do not need to be copied to other segments at all.

4. Low-level Implementation

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

4.1 Application Programming Interface

Listing 1 Application programming interface

```
void stm_start_transaction()
void stm_commit_transaction()
void stm_become_inevitable()
void stm_read(object_t *SO)
void stm_write(object_t *SO)
object_t *stm_allocate(size_t size_rounded)
STM_PUSH_ROOT(object_t *SO)
STM_POP_ROOT(object_t *SO)
```

Our TM system is designed as a C library that covers all aspects around transactions and object memory management. It is designed for object-oriented dynamic language VMs as a replacement for the GIL. The library exposes the functions in Listing 1 as its interface for use by a dynamic language interpreter.

`stm_start_transaction()` starts a transaction in the current thread. Internally, it uses `setjmp()` to remember where the transaction started in case of an abort. `stm_commit_transaction()` tries to commit the current transaction, aborting it otherwise. Similarly, `stm_become_inevitable()` tries to make the current transaction inevitable, waiting or aborting the current one if there is already an inevitable transaction. `stm_read()`, `stm_write()` perform a read or a write barrier on the object given as the argument, and `stm_allocate()` allocates a new object with the specified size.

`STM_PUSH_ROOT()` and `STM_POP_ROOT()` push and pop objects onto the shadow stack[2] of the current thread. References to objects have to be saved using this stack locally, and conservatively around calls that may cause a GC cycle to happen, and also while there is no transaction running. Otherwise these references may not be valid any more because the object moved or was discarded as garbage. In this simplified Application Programming Interface (API), only `stm_allocate()`, `stm_become_inevitable()` and `stm_commit_transaction()` require saving object references.

In the following sections, whenever we use SO, we go through the address translation to get to the actual contents of an object. This is also signified by the type `object_t`. This type is special as it causes the compiler[3] to make all accesses through it relative to the $\%gs$ register. With exceptions, nearly all accesses to objects managed by the TM system use this type so that the CPU will translate the reference to the right version of the object.

4.2 Replacing the GIL

With this simple API, replacing the GIL is just a matter of starting a new transaction when we would usually acquire the GIL, and committing the current transaction when we normally release the GIL. It would be wasteful to do this between every single bytecode instruction. We thus depend on a simple heuristic that estimates how expensive aborting and retrying the current transaction is based on how much memory was allocated in the current transaction. If we deem it expensive enough, we commit and start a new transaction *as soon as possible*. Deciding on the length of a transaction is an interesting challenge in and of itself.

It may not be possible to immediately break the transaction, as we could be executing an atomic block. Atomic blocks are currently implemented as guaranteeing that the whole block's execution takes place in a single transaction.

Another issue is the support of irreversible operations, like calls to external functions or general input / output. If Python allows it, we commit the transaction before and start a new one after these operations (the same way as the GIL is released around thread-safe external functions). Within an atomic block, however, the transaction is unbreakable, and we need to turn the transaction

[2] 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. [14]

[3] We use Clang 3.5 with patches to deal with bugs in its "address-space 256" feature. Patches are available from authors until inclusion into the official clang.

inevitable so that it cannot abort and strong isolation is guaranteed at all times.

4.3 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 sharing-segment is essentially read-only (except during commit) and holds all pages shared between the other segments.

The next step involves using `remap_file_pages()`, a Linux system call[4], 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 privatise a few areas again right away. These areas are illustrated in Figure 3 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 for bookkeeping.

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

Nursery: This private area contains all the freshly allocated objects (*young objects*) of the current transaction. The GC uses bump-pointer allocation in this area to allocate objects of the first generation. To make sure that only objects of the current transaction reside in this space, a collection of the first generation happens at the end of every transaction (see section 4.5).

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 be allocated in the nursery.

Note, since the above configuration is currently specified at compile time, all these areas are at offsets inside the segments known to the compiler. This makes some checks very efficient, e.g. checking if an object resides in the nursery only requires comparing its SO to the static boundaries of the nursery. It is possible that we want some parameters to be configurable at startup or even during the runtime. In that case we can still use a compile-time specified maximum so that these checks are still efficient. E.g. limiting the maximum amount of memory available to the application statically to a few terabytes is fine because it corresponds to virtual memory; the real physical memory is assigned on-demand by the operating system.

4.4 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 needs to acquire a segment. It may have to wait until another thread finishes its transaction and releases its segment. Fairness is not guaranteed yet, as we simply assume a fair scheduling policy in the operating system when waiting on a lock.

[4] remap_file_pages() can also be done with more mmap() calls in a roughly portable way, but the former is more efficient on Linux.

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 to maximise the effectiveness of caches and the page privatisation.

4.5 Garbage Collection

Garbage collection plays a big role in our TM system. The GC is generational and has two generations: the *young* and the *old* generation. It is optimised for dynamic languages with high allocation rates.

The **young 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 a thread's nursery into the old object space, which can be done without stopping other threads. This happens 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 the write barrier that the object still belongs to the current transaction. Other and later transactions will treat them like regular *old* objects.

The **old 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 during major collections. This is another example where the STM-GC cooperation makes sense. We regard private pages as additional memory that may be collected. Pages should usually stay private for some time, but at some regular intervals, we still want to optimistically re-share pages in case they actually stay unmodified in the future. Major collections are allowed to do that because all threads are synchronised, and we can re-share all private pages that do not currently contain an object in some transaction's write set.

As seen in the API (section 4.1), we use a *shadow stack* [14] 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()`.

4.6 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. In other STM systems, it also resolves an object reference to a private copy, but since the CPU performs our address translation on every object access efficiently, we can avoid this extra overhead.

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 4.3.

This area can be seen as a continuous, segment-local array of bytes that is indexed with an object's reference (SO) divided by 16. So that each object has its own read marker, all objects have a size of at least 16 bytes. Otherwise there could be false

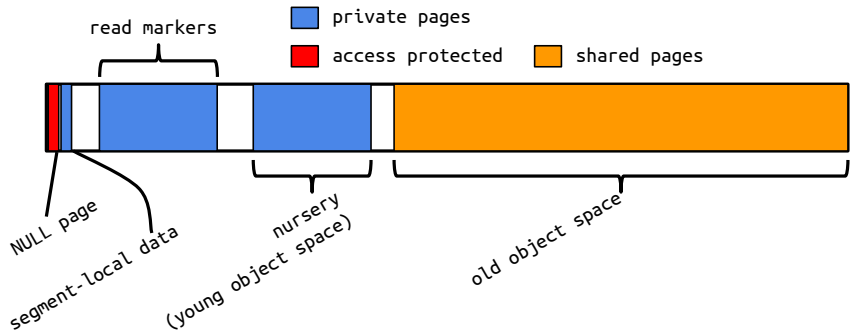

Figure 3. Segment Layout

conflicts when reading from two adjacent objects that share a single marker. 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 shown in Listing 2 is easily optimisable for compilers as well as perfectly predictable for CPUs. Additionally, the read barrier is not constrained to execute before the actual read – both the compiler and the CPU can reorder or group them freely between potential safe points for additional performance.

Listing 2 The complete read barrier

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

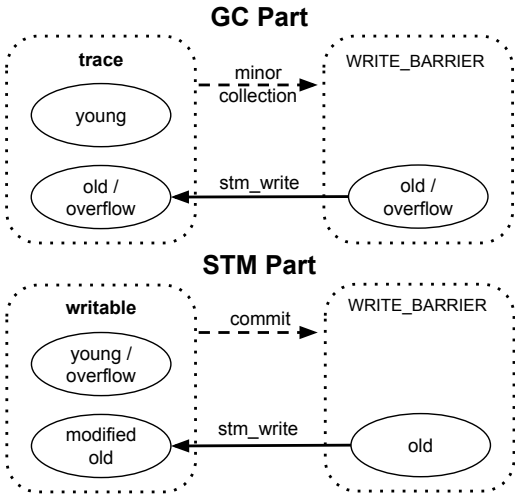

Figure 4. Object states handled by the write barrier

4.7 Write Barrier

The job of the write barrier is twofold: it serves as a write barrier for the garbage collector (GC) and the STM system. These two jobs are depicted in Figure 4. The GC needs to keep track of objects that may contain references to young, movable objects. These objects need to be traced during minor collections to update references to moved objects. Young objects automatically get traced if the GC ascertains that they survive the collection by being reachable from traced objects. So far, this is how generational GCs generally work.

But the STM system also needs to record objects in the write set, acquire write locks, and privatise their pages (copy-on-write). It only needs to do this for old objects from previous transactions, as any other object is not known to other transactions anyway.

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 through its SO 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 (in this case, the flag is added again on all objects, including new overflow objects). The flag is never set on young, freshly allocated objects. This fast path covers all cases for the GC and for the STM system (see Listing 3).

Listing 3 Write barrier fast path

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

Listing 4 Slow path of the write barrier

```
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
```

The **slow path** is shown in Listing 4. For each object, this code is called at most once between two minor collections. 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`). Any reference we write to this object will be kept alive and up-to-date during minor collections.

The check for `is_overflow_obj()` looks at the `overflow_number` and tells us if the object was actually created in this transaction. In that case, we do not need to execute the following *STM part*. We especially do not need to privatise its pages since no other transaction knows about these overflow objects. Even if they reside in non-private pages, it is guaranteed that no other transaction can have a reference to them.

Up to here, we see that all cases of the GC shown in Figure 4 are handled. What is left is to handle the cases of the STM system that need to make old objects writable.

For the *STM part*, we first perform a read barrier on the object. We then try to acquire its write lock. `write_locks` is a simple, *global* array of bytes that is indexed with the SO of the object divided by 16. Note, "global" here really means it is a single array with data for all segments, there is no address translation going on to access its elements contrary to e.g. the read markers array. If we already own the lock, we are done. If someone else owns the lock, we perform write-write contention management that aborts either us or the current owner of the object. If the lock is not owned so far, and if we succeed in acquiring it using an atomic `cmp_and_swap`, we need to add the object to the write set (a simple list called `modified_old_objects`). We then privatise the page or pages where it resides (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 or we start the next transaction (remember: we always perform a minor collection as part of each commit anyway).

Note that we have three kinds of objects: *young, overflow* and *old* objects. Young and overflow objects were created in the same transaction; old objects always come from previous transactions. The close integration between STM and the GC allows us to use this information effectively: young objects do not trigger the slow path at all, overflow objects only need to be handled by the GC part of the write barrier once between collections, and only the old objects need to be handled by the STM part *once per transaction*. The privatisation step in particular is done very rarely because the pages stay private until the next major collection where they may get re-shared. This keeps the overhead of the write barrier very low in the common cases.

4.8 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 initialised by a `setjmp()` in `stm_start_transaction()`. Increasing the `read_version` for the next transaction is also done there.

4.9 Commit

Committing a transaction needs a bit more work. First, we synchronise all threads so that the committing one is the only one running and all the others are waiting in safe points. 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 perform contention management to either abort us or the other transaction, or to simply wait a bit (see Section 4.11).

Note that synchronising all threads is an important design choice we made to support our very cheap read barriers. Since they now consist only of a single unconditional write to some memory location, all threads need to be stopped while we detect conflicts. This is a trade-off that works best if the number of concurrent threads is small.

After verifying that there are no conflicts anymore, we copy all our changes (i.e. objects in our write set) to all other segments, including the sharing-segment. This is safe since we synchronised 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 counter, so that it stays globally unique for each transaction. Increasing the (local) `read_version` is then done at the start of a new transaction.

4.10 Thread Synchronisation

A requirement for performing a commit is to synchronise all threads so that we can safely update objects in other segments. To make this synchronisation fast and cheap, we do not want to insert an additional check regularly to see if synchronisation 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 in Listing 5.

Listing 5 Function to allocate objects

```
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 bump-pointer 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 to free up space in the nursery.

If we want to synchronise all threads, we can rely on this check being performed regularly. So what we do is to set the `nursery_end` to some small number in all segments that we want to synchronise. 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.

Note, this works well for interpreters that really allocate things all the time. In the presence of an optimising just-in-time compiler, some loops may indeed not allocate anything and therefore still need the insertion of a safe-point check.

4.11 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 increase the chance of long transactions succeeding
- to support *inevitable* transactions, we always prefer them to others since they cannot abort (similar to [12])

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.

5. Evaluation

We evaluate our system in a Python interpreter called PyPy and compare to CPython.

PyPy (version 2.3) is an implementation of an interpreter for the Python language. It has a special focus on speed and provides a just-in-time (JIT) compiler to speed up applications running on top of it. We compare between normal PyPy using a GIL and a PyPy with STM. All code is available at [8].

CPython (version 2.7.6) is the reference implementation of the Python language. It is the most widely used interpreter for this language. The implementation uses a GIL for synchronisation in multi-threaded execution and it does not feature a JIT compiler.

Here, we will not go into detail about the integration of our STM system with PyPy's JIT. In fact, we will disable it for all benchmarks except those in Section 5.4. The STM-JIT integration is currently still incomplete and not tested much. The JIT-less interpreter provides a much more consistent environment for the STM system, so we remove some unknown variables by disabling it. Furthermore, we think the interpreter-STM evaluation is more relevant at this stage as the results can be more directly applied to other similar interpreters. PyPy's JIT [1], however, is quite unique as it is a JIT tracing the interpreter instead of the interpreted language itself. This demands its own thorough evaluation, which is out of the scope of this paper.

We performed all benchmarks on a machine with an Intel Core i7-4770 CPU @3.40GHz (4 cores, 8 threads) with disabled Hyper-Threading and Turbo Boost for less variation in the results. There are 16 GiB of memory available and we ran them under Ubuntu 14.04 with a Linux 3.13.0 kernel. The STM system was compiled with a number of segments $N = 4$ and a maximum amount of memory of 1.5 GiB (both configured at compile time).

For each point in the plots, we took 5 measurements and report the arithmetic mean with the standard deviation as error bars. Average speedup numbers are calculated using the geometric mean. For the JIT benchmarks, we first let the JIT warm up by doing a few additional iterations before the actual measurement.

5.1 Scaling

To asses how well the STM system in combination with a Python interpreter scales on its own (without any real workload), we execute the loop in Listing 6 on 1 to 4 threads on the PyPy interpreter with STM and without a JIT. There are very few allocations or calculations in this loop, so the main purpose of this benchmark is simply to check that there are no inherent conflicts in the interpreter when everything is thread-local. We also get some idea about how much overhead each additional thread introduces.

Listing 6 Dummy workload

```
def workload():
    i = 20000000
    while i:
        i -= 1
```

The STM system detected no conflicts when running this code on 4 threads. For the results in Figure 5, we normalised the average runtimes to the time it took on a single thread. From this we see that there is some additional overhead introduced by each thread (9.1% for all 4 threads together). Every thread adds some overhead because during a commit, there is one more thread which has to reach a safe point. Additionally, conflict detection needs to check for conflicts in all concurrently running transactions.

While not ideal, we think that 9.1% is acceptable on four threads. In terms of throughput, 4 threads reach 3.67× the iterations per second of a single thread.

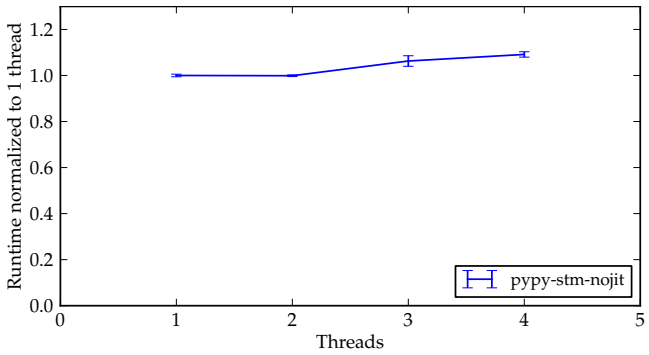

Figure 5. Scalability of the STM system

5.2 Small-Scale Benchmarks

For the following sections we use a set of 8 small benchmarks available at [19]. There are, unsurprisingly, not many threaded applications written in Python that can be used as a benchmark. Since until now, threading was rarely used for performance reasons because of the GIL, we mostly collected small demos and wrote our own benchmarks to evaluate our system:

- *btree* and *skiplist*, which are both inserting, removing, and finding elements in a data structure from multiple threads
- *threadworms*, which simulates worms walking on a grid in parallel and checking for collisions with each other
- *miller-rabin*, *mandelbrot*, *raytrace*, *richards*, and *mersenne*, which all perform some mostly independent computations in parallel (embarrassingly parallel).

We use atomic blocks as the primary mechanism to synchronise access to shared data structures in the first three benchmarks. For the embarrassingly parallel benchmarks, we only need explicit synchronisation in some small isolated part. These atomic blocks are simulated with a single, global lock when running on top of GIL-supported interpreters. With our STM system, they map to transactions that will execute optimistically in parallel.

5.3 Non-JIT Benchmarks

First we run our benchmarks on three different interpreters: CPython (GIL), PyPy with STM, and PyPy with the GIL (all without a JIT). The results are shown in Figure 6.

As expected, GIL-supported interpreters do not scale with the number of threads. They even become slower because of the overhead of thread-switching and GIL handling (see [9] for a detailed analysis).

On 4 cores, PyPy using our STM system (*pypy-stm-nojit*) scales in all benchmarks to a certain degree. *mersenne* is a bit of a special case, since it is easily parallelisable but a few of the parallel computations dominate the runtime and cannot be made faster easily by parallelisation. This is why the benefit of adding more threads than 2 is diminishing.

pypy-stm-nojit scales best for the embarrassingly parallel benchmarks ($avg = 2.7\times$ speedup over 1 thread) and a little less for the others ($avg = 2.5\times$ speedup over 1 thread). The reason for this difference is that in the former group there are no real, logical conflicts – all threads do independent calculations. STM simply

replaces the GIL in those programs. In the latter group, the threads work on a common data structure and therefore create much more conflicts, which limits the scalability. Here we make active use of the STM-supported atomic blocks to synchronise access to the data structure. The hope is that the STM system is still able to parallelise, even if we use the atomic blocks in a coarse-grained way. While less than the other benchmarks, we indeed see some speedup going from 1 to 4 threads.

Looking at the average overhead on a single thread that is induced by switching from GIL to STM, we see that it is $\approx 40.1\%$. The maximum in richards is 65.5%. In all benchmarks *pypy-stm-nojit* beats *pypy-nojit* already on two threads, despite of this overhead. The achieved speedup comparing the fastest runtimes of STM to the single-threaded GIL execution of *pypy-nojit* is between $1.11\times$ and $2.55\times$.

Still, *pypy-stm-nojit* barely beats CPython's *single-thread* performance. However, for programs that need concurrency in CPython and that use threads to achieve this, it also makes sense to look at the overhead induced by the GIL on multiple threads. From this perspective, the STM implementation beats CPython's performance in all benchmarks.

Since PyPy comes with a JIT [1] to make it vastly faster than CPython, we will now look at how well STM works together with it.

5.4 JIT Benchmarks

We would like to regard enabling the JIT as a simple performance enhancement, but that is not what happens in reality. First, since the JIT [1] bases its decision of what to compile on runtime profiling, running in multiple threads, it may compile different things in each run because of the non-deterministic thread-scheduling of the operating system (OS). Second, it is able to perform some advanced optimisations. Because compilation is already non-deterministic, so are these optimisations. And third, we did not have enough time to optimise integration with STM so that the JIT exposes the overhead of STM more by speeding up all the rest. For these reasons, the following results should be regarded as a preliminary outlook.

The speedups from simply enabling the JIT in these benchmarks range from $10 - 50\times$. This is why we had to do without CPython here, since it would be much further up in the plots. Also, to make jitting code worthwhile, we increased the input size of all benchmarks to get reasonable execution times (denoted with "(large)" in benchmark names).

We also ran these benchmarks on Jython (v2.7b1 from [6]), an implementation of Python on top of the Java Virtual Machine (JVM). Instead of a GIL, this interpreter uses fine-grained locking for synchronisation. Even though it can use the JVM's JIT compiler, its performance in these benchmarks is behind PyPy's by a factor of $5 - 10\times$. Because of that it does not make sense to include it in this evaluation. In the future, we would like to do an extensive comparison between the STM approach and fine-grained locking. It is out of the scope of this paper to do this thoroughly.

The results are presented in Figure 7. We see that the performance gains of STM are a bit less reliable with the JIT. The slowdown factor for switching from GIL to STM ranges around $1 - 2.5\times$, and we beat the GIL's single-thread performance in 5 out of 8 benchmarks.

We see that generally, the group of embarrassingly parallel benchmarks scales best. The other three benchmarks scale barely or not at all with the number of threads. The reason for this is likely again the conflicts in the latter group. Right now, because the code runs much more quickly with the JIT than without, it has the effect of making the transactions logically longer. This

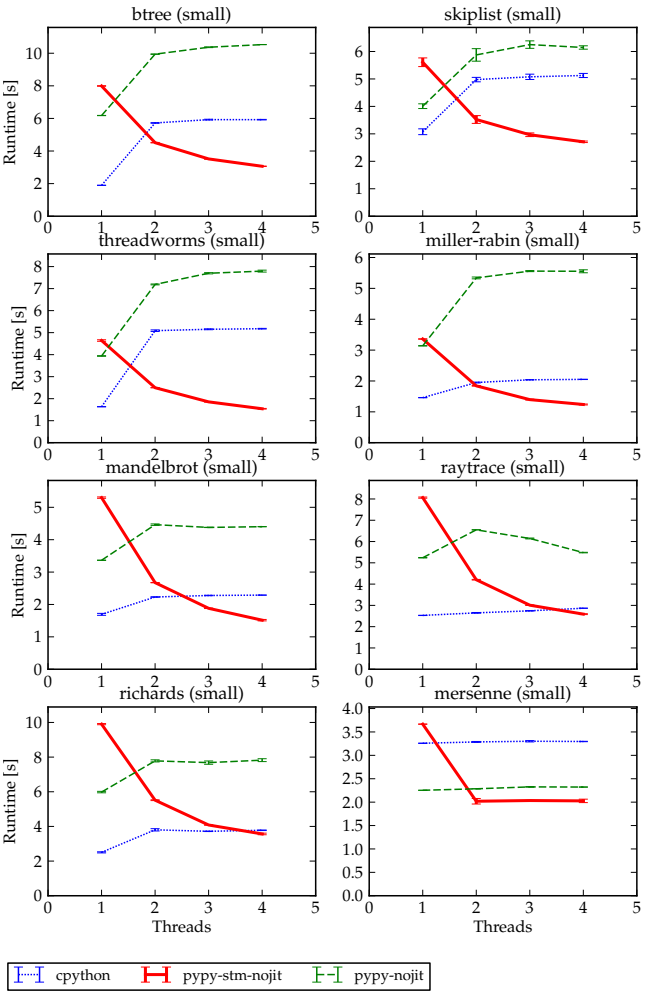

Figure 6. Comparing execution time between interpreters without a JIT and using small input sizes

increases the likelihood of conflicts between them and therefore limits scalability even more than in the no-JIT benchmarks.

Overall, PyPy without STM is around $2\times$ slower than CPython (*cpython* vs. *pypy-nojit*). Enabling its JIT allows it to outperform CPython by a huge margin. We see the same kind of speedup on PyPy with STM when enabling the JIT. This means that STM does not generally inhibit the JIT from optimising the programs execution, which is already a very important result on its own. We also see that for some benchmarks, STM is already able to give additional speedups compared to just the JIT-induced acceleration. This looks very promising and improving this situation even more is the next step for reaching the goal of parallelising Python.

6. Related Work

There have been several attempts at removing the GIL using TM. We ourselves did a few earlier attempts with more conventional STM systems [15]. Because of the huge overhead we encountered, the result was mostly useless on ≤ 4 CPU cores. Since this could be considered the main operating conditions of dynamic languages, it was never as useful as we wanted it to be.

Other attempts using HTM [20, 24, 25] show promising results, especially the very recent [20] that examines the current generation

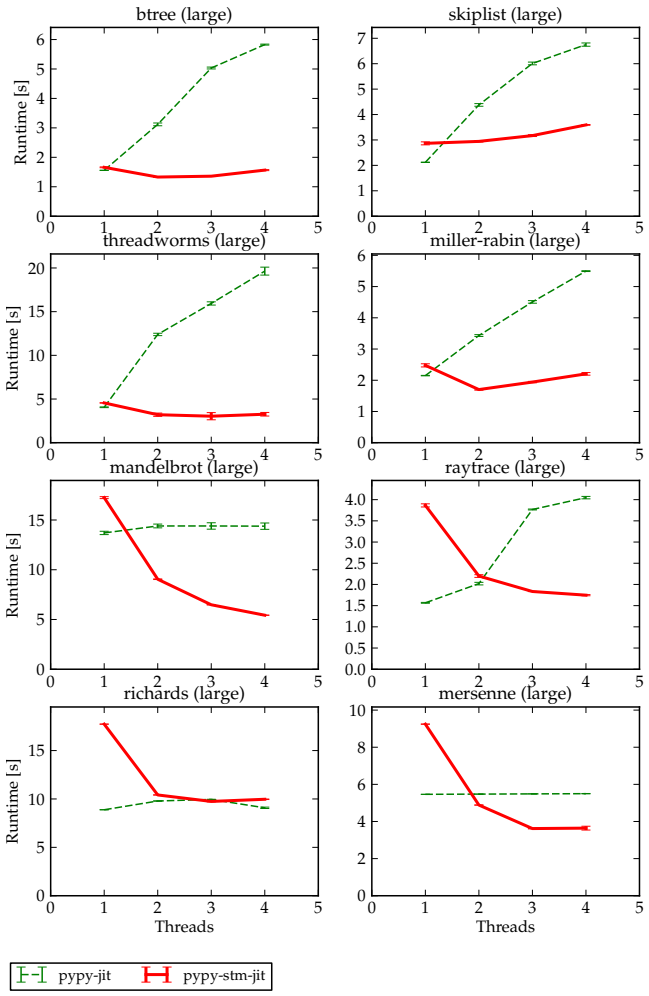

Figure 7. Comparing execution time between interpreters with JIT and large input sizes

of Intel CPUs. However, they did a simplification by essentially disabling the GC during the benchmarks. This may be valid for the hand-tuned allocations in their interpreter. For us the GC is essential, and because it is a moving GC, it unfortunately touches a lot more memory and thereby reaches the HTM capacity much faster. This, and the fact that HTM's capacity limits do not allow the implementation of arbitrary-sized atomic blocks, let us to believe that STM is the right choice at the moment.

There are multiple attempts at removing the limits of HTM by means of *virtualising* HTM [16, 17]. Indeed this may also be the future direction of our research. So far, some of them depend on additional hardware features. Or in the case of [17], they have the assumption that most transactions fit the HTM capacity and that virtualisation is only a fallback. Our use case, however, features arbitrarily long transactions, which is why this assumption first has to be verified again in this setting.

From the other end, there also have been attempts at lowering the overhead of STM [21, 27] especially for a low number of CPU cores. [27] works well for some workloads, in the case of a dynamic language VM, however, we have to assume any possible workload. [21] uses a compile-time approach to select from multiple optimised STMs. Their partitioned FastLane STM seems to provide good performance already on two threads using master and helper threads. It would be interesting to see if its design also performs well in our use case.

Similarly to us, [18] use memory management techniques to provide isolation between transactions. They focus on providing strong atomicity between transactional and non-transactional code. We, however, use inevitable transactions to provide strong atomicity for non-reversible operations and otherwise do not have to deal with non-transactional code. Additionally, while they use page fault handlers to detect conflicts, we use traditional write barriers. There are only some superficial similarities between the approach presented here and theirs.

7. Conclusions

As multi-core processors continue to increase the number of cores per processor, it is important that dynamic languages like Python embrace parallelism. We report here an effort to replace the Global Interpreter Lock (GIL) used in many Python implementations with transactional memory (TM). The key factor for success is a close integration of a software TM (STM) and the garbage collector to provide atomicity, synchronisation, and memory management in a unified way. The STM system supports atomic blocks as a user abstraction and thereby provides a scalable synchronisation mechanism to Python applications.

The solution presented here is based on a software TM system developed as an independent and reusable C library for integration in dynamic language interpreters. As the STM is a vital part of the interpreter, good performance is essential. We present an interesting way to combine the existing memory segmentation and virtual memory features of current CPUs to lower the overhead of STM. This software implementation reduces dramatically the cost of transactional memory and at the same time provides an opportunity to integrate the STM system with garbage collection. This implementation leverages the processor's memory segmentation hardware, which has lately not received much attention. Unfortunately, processor architects seem to undervalue this hardware resource – in the evolution from the x86 to the x86-64 architecture, 4 out of 6 segment registers have been removed. This development potentially threatens the STM strategy outlined here and we want to remind the architecture community that this unit can support interesting implementations – a viewpoint that has also been expressed by other researchers (see also [34]).

The early results presented here are very encouraging. STM as a simple GIL replacement scales well and yields an average speedup of $2.7\times$ for embarrassingly parallel workloads on 4 threads. When also used for synchronisation in the form of atomic blocks, the average speedup still reaches $2.5\times$.

To generally outperform the best-performing Python systems (Jython, PyPy), integration of the STM-based approach with a JIT compiler is necessary. Our early results of this integration suggest that there is no inherent incompatibility between STM and PyPy's JIT. Once the implementation matures, the approach outlined here serves not only as a simple GIL replacement but also provides a way forward towards a parallel programming model for Python.

Acknowledgments

We would like to thank the following people for their valuable input and the many fruitful discussions: Carl F. Bolz, Michael Faes, Pascal Felber, Maciej Fijalkowski, Patrick Marlier, and Aristeidis Mastoras.

References

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. "Tracing the meta-level: PyPy's tracing JIT compiler." *Proc. 4th workshop on the Implementation, Compilation, Optimization of*

Object-Oriented Languages and Programming Systems (ICOOOLPS '09), 18-25

[2] Kevin Millikin, Florian Schneider. 2010. "A New Crankshaft for V8." `http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html`

[3] IonMonkey from Mozilla. 2014. `https://wiki.mozilla.org/IonMonkey/Overview`

[4] Remigius Meier, Armin Rigo. 2014. "A Way Forward in Parallelising Dynamic Languages." In *Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE* (ICOOOLPS '14)

[5] CPython. `www.python.org`

[6] The Jython Project, `www.jython.org`

[7] IronPython. `www.ironpython.net`

[8] PyPy Project. `www.pypy.org`

[9] David Beazley. "Understanding the Python GIL." *PyCON Python Conference*. Atlanta, Georgia. 2010.

[10] Tim Harris, James Larus, and Ravi Rajwar. 2010. "Transactional memory." *Synthesis Lectures on Computer Architecture 5.1*

[11] Rachid Guerraoui and Michal Kapalka. 2008. "On the correctness of transactional memory." In *Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (PPoPP '08), 175-184

[12] Colin Blundell, E. Christopher Lewis, and Milo Martin. 2006. "Unrestricted transactional memory: Supporting I/O and system calls within transactions."

[13] Michael F. Spear and Michael Silverman and Luke Daless and Maged M. Michael and Michael L. Scott. 2008. "Implementing and exploiting inevitability in software transactional memory." In *Proc. 37th IEEE international conference on parallel processing* (ICPP '08), 59-66

[14] Fergus Henderson. 2002. "Accurate garbage collection in an uncooperative environment." In *Proc. 3rd International Symposium on Memory management* (ISMM '02), 150-156

[15] Armin Rigo, Remigius Meier. 2013. "Update on STM." `morepypy.blogspot.ch/2013/10/update-on-stm.html`

[16] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. "Virtualizing Transactional Memory." In *Proc. 32nd annual International Symposium on Computer Architecture* (ISCA '05), 494-505

[17] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. 2006. "Tradeoffs in transactional memory virtualization." In *Proc. 12th international conference on Architectural support for programming languages and operating systems* (ASPLOS XII), 371-381

[18] Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. "Transactional memory with strong atomicity using off-the-shelf memory protection hardware." In *Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '09), 185-196

[19] PyPy benchmarks repository. 2014. Revision fd2da4da8f33. `bitbucket.org/pypy/benchmarks`

[20] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 2014. "Eliminating global interpreter locks in Ruby through hardware transactional memory." In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '14), 131-142

[21] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. 2013. "FastLane: improving performance of software transactional memory for low thread counts." In *Proc. 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '13), 113-122

[22] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2011. "Why STM can be more than a research toy." *Commun. ACM* 54, 4 (April 2011), 70-77.

[23] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. "Software transactional memory: why is it only a research toy?." *Commun. ACM* 51, 11 (November 2008), 40-46.

[24] Nicholas Riley and Craig Zilles. 2006. "Hardware transactional memory support for lightweight dynamic language evolution." *In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (OOPSLA '06), 998-1008

[25] Fuad Tabba. 2010. "Adding concurrency in Python using a commercial processor's hardware transactional memory support." *SIGARCH Comput. Archit. News 38*, 5 (April 2010), 12-19.

[26] Pascal Felber and Torvald Riegel and Christof Fetzer and Martin Süßkraut and Ulrich Müller and Heiko Sturzrehm. 2007. "Transactifying applications using an open compiler framework." *TRANSACT*, August (2007): 4-6.

[27] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. 2010. "Transactional mutex locks." In *Proc. 16th international Euro-Par conference on Parallel processing: Part II* (Euro-Par'10), Pasqua D'Ambra, Mario Guarracino, and Domenico Talia (Eds.). Springer-Verlag, Berlin, Heidelberg, 2-13.

[28] L. Lamport. 1979. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Trans. Comput.* 28, 9 (September 1979), 690-691.

[29] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. "A study of transactional memory vs. locks in practice." In *Proc. 23rd annual ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA '11), 43-52

[30] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. "Is transactional programming actually easier?." *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '10), 47-56

[31] Tim Harris and Keir Fraser. 2003. "Language support for lightweight transactions." In *Proc. 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (OOPSLA '03), 388-402.

[32] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2008. "Composable memory transactions." *Commun. ACM* 51, 8 (August 2008), 91-100.

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." *SIGARCH Comput. Archit. News* 36, 1 (March 2008), 329-339.

[34] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. "Native Client: a sandbox for portable, untrusted x86 native code." *Commun. ACM* 53, 1 (January 2010), 91-99.