Parallel Virtual Machines with RPython

Remigius Meier
Department of Computer Science
ETH Zurich
Zürich, Switzerland

Armin Rigo
PyPy Project
www.pypy.org
Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich
Zürich, Switzerland

Abstract

The RPython framework takes an interpreter for a dynamic language as its input and produces a Virtual Machine (VM) for that language. RPython is being used to develop PyPy, a high-performance Python interpreter. However, the produced VM does not support parallel execution since the framework relies on a Global Interpreter Lock (GIL): PyPy serialises the execution of multi-threaded Python programs.

We describe the rationale and design of a new parallel execution model for RPython that allows the generation of parallel virtual machines while leaving the language semantics unchanged. This model then allows different implementations of concurrency control, and we discuss an implementation based on a GIL and an implementation based on Software Transactional Memory (STM).

To evaluate the benefits of either choice, we adapt PyPy to work with both implementations (GIL and STM). The evaluation shows that PyPy with STM improves the runtime of a set of multi-threaded Python programs over PyPy with a GIL by factors in the range of $1.87\times$ up to $5.96\times$ when executing on a processor with 8 cores.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords global interpreter lock, transactional memory, Python, RPython, parallelism, virtual machine, dynamic language

1. Introduction

The RPython framework aids in the development of high-performance Virtual Machines (VM) for dynamic languages. The framework has the goal of separating language specification from implementation aspects, making the latter reusable among VMs. The interpreter serves as the specification of the language, and the framework combines the interpreter with other components that handle various implementation aspects. Given an interpreter for a language, the framework's toolchain produces a complete VM that, in addition to that interpreter, has a garbage collector and a just-in-time compiler.

The most prominent example of a VM developed with the RPython framework is PyPy, a Python VM. The PyPy VM is one of the fastest Python implementations available. However, it cannot execute any Python code in parallel, even if a program makes explicit use of threads. For the last decade, processor performance has primarily increased through the availability of multiple execution engines (cores), but Python programs running on PyPy cannot benefit from this development and their performance is, thus, severely limited.

The source of this limitation lies within the RPython framework. Because the framework takes an executable interpreter as its input, that interpreter needs an execution model that allows parallel execution. An execution model defines the concurrent units of a language and the ways these units can interact with each other. RPython's execution model is a threading model with shared memory, but interaction and execution of threads is defined with a Global Interpreter Lock (GIL). Each thread is allowed to progress only after it exclusively acquired the GIL, and a thread must explicitly yield the GIL to other threads. Hence, the execution is serialised over all threads. The only workaround for Python programs to benefit from parallel platforms is to use multiple VMs in separate processes with explicit communication channels.

Because the source of this limitation is within the RPython framework itself and, hence, affects all VMs developed with that framework, the goal is to design a new execution model for RPython that allows for parallel execution with the same semantics. Parallel execution is only possible by removing the current model's reliance on the GIL. However, the GIL provides one strong and useful guarantee for protected code: atomic execution.

Atomic execution can be supported by other means such as fine-grained locking. IronPython [1] and Jython [2], two

DLS'16, November 1, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4445-6/16/11...
http://dx.doi.org/10.1145/2989225.2989233

Python implementations, avoid the GIL with a lot of duplicated effort and manual work. However, the RPython framework requires a solution that is reusable among different interpreters, so we describe an alternative solution based on transactional memory (TM).

TM provides the abstraction of a transaction, which ensures atomic execution for a group of operations and its memory accesses. There have been previous attempts to replace the GIL with hardware TM [3; 4; 5], but these efforts were plagued by various implementation problems and, to some extent, imposed restrictions on the interpreter. Such restrictions violate the goal of separating language specification from implementation aspects, a crucial factor if we want to use the RPython framework for different languages and interpreters.

In this paper, we first describe the issues of the current execution model of RPython with the example of PyPy, the Python interpreter. We detach the model from the GIL by introducing the concept of quantised atomicity [6], the division of a concurrent execution into atomic quantum regions (quanta). A quantum is an abstraction that helps with specifying concurrency semantics of an interpreted language and that makes concurrency control an exchangeable component within RPython's execution model. We then introduce a new concurrency control component that uses software TM (STM) to supports parallel execution of quanta.

As a small case study, we report on the issues encountered with the PyPy interpreter by the introduction of STM. We then compare the two versions of PyPy, one with a GIL-based component to handle concurrency and one with an STM-based component, on a set of multi-threaded Python programs. This case study demonstrates that parallel Python programs obtain real benefits through parallel execution, and that RPython can be used to construct a high-performance VM that supports parallelism.

In short, the contributions of this paper are the following:

- The introduction of a new, parallel execution model for RPython that is freed from the GIL legacy with the help of quantised atomicity;
- a description of using the new quantum abstraction to specify concurrency semantics in interpreters;
- a description of a parallel implementation of the new model using STM;
- a case study of porting PyPy to use the new STM-based component;
- a performance comparison between two versions of the PyPy VM, one based on the GIL, and one using the STM component on a set of multi-threaded Python programs.

2. The PyPy Project

The *PyPy project* aims to deliver a high-performance *Virtual Machine* (VM) for the language Python. The parts most relevant for this discussion are the *PyPy interpreter* and the *RPython framework*. The former is a Python interpreter that uses the latter to generate a Python VM.

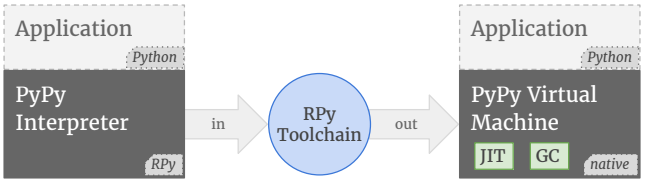

Figure 1. The RPy toolchain translates the PyPy interpreter to a PyPy VM and mixes in additional VM components.

2.1 The RPython Framework

A VM is an implementation of a language. It provides a (portable) execution environment for programs written in that language. A VM usually contains an interpreter as well as other components such as a garbage collector or a just-in-time compiler. These components cooperate and provide the execution environment. For example, the interpreter executes the program while a garbage collector reclaims memory and a just-in-time compiler specialises the program for the concrete platform.

The RPython framework is a flexible framework for developing such VMs for dynamic languages. The framework consists mainly of the language *RPython*, libraries and components written in RPython, and the *RPython toolchain*. "RPython" stands for "Restricted Python", a statically typed subset of Python. For better textual differentiation from the word "Python", we henceforth use the short form *RPy*.

RPy is an object-oriented language that allows its programs to be translated to efficient C code, and from there to a native executable. As such, RPy is not too different from many other statically compilable languages. Where it differs greatly, however, is the translation process done by the RPy toolchain.

The RPy toolchain's goal is to separate language specification from implementation aspects. An interpreter written in RPy serves as an executable specification of the interpreted language, and implementation concerns are encapsulated within separate, reusable components. The RPy toolchain mixes the two aspects together in a controllable way to produce the PyPy VM.

Figure 1 depicts this process for the PyPy interpreter: The PyPy interpreter, which interprets the language Python and is itself written in RPy, serves as the specification of the language Python and is given as input to the RPy toolchain. The RPy toolchain translates that input, mixes in additional VM components, and produces a full, native VM — the *PyPy VM*.

Using an interpreter written in RPy as its input, the RPy toolchain allows the choice between multiple implementation strategies to be a parameter of the translation. For example, the PyPy interpreter can be translated using several garbage

collection strategies. The toolchain even goes as far as giving a translation choice for adding a just-in-time compiler to the VM [7]. This ability to make such advanced VM components reusable for different interpreters makes the RPy framework uniquely suited for developing VMs for dynamic languages.

2.2 Specification of Sequential Semantics

To illustrate how an interpreter specifies the semantics of the interpreted language, and how the RPy toolchain separates the language semantics (expressed by the interpreter) from implementation aspects, we look at a small Lisp-like language, called *Duhton*, and examine a small part of its interpreter.

Duhton supports the special form `while` that repeatedly checks a condition and evaluates its arguments if that condition is considered true. Listing 1 is an example of using that special form to repeatedly increase a counter until it reaches 10.

Listing 1 A while loop in Duhton counting from 0 to 10

```
(setq n 0)
(while (< n 10)
  (setq n (+ n 1)))
```

The semantics of `while`, i.e., the meaning of the construct, is a concern separate from, e.g., how garbage collection reclaims memory from temporary values created by the expression `(+ n 1)`. The semantics is also independent of how a just-in-time compiler (JIT) expresses the above loop in native code. RPy abstracts these implementation aspects and allows an interpreter for Duhton to specify the semantics of `while` in the following way:

Listing 2 `while` implementation in RPy

```
def duhton_while(head, tail, frame):
  while True:
    jit_merge_point(...)
    if not head.eval(frame).is_true():
      break
    duhton_progn(tail, frame)
```

Listing 2 shows an RPy function that takes the `head` and `tail` of the special form `while` and evaluates them in the current `frame`. The evaluation is expressed with a loop that repeatedly evaluates the condition in `head`. If the evaluation yields `true`, the body in `tail` gets evaluated using another function `duhton_progn`, and otherwise the loop exits.

The function `duhton_while` expresses the semantics of the special form `while` without managing intermediate values constructed by, e.g., `head.eval(frame)`. The interpreter can assume that there is some form of garbage collector that reclaims the memory. Further, there is `jit_merge_point`, a hint to a possibly mixed-in JIT. That hint is needed to communicate to the JIT where loops start and close in Duhton, which is right before evaluating the condition of Duhton's `while`. Additionally, the choice of a garbage collector and whether to use a JIT or not is made with a translation option passed to the RPy toolchain.

However, the above examples illustrate only how the interpreter specifies the *sequential* semantics of Duhton in RPy, but a language with concurrency also has a *concurrency* semantics. For Python, which supports concurrent threads, we thus need a way to implement its concurrent execution model and specify its concurrency semantics in RPy. But first, an accurate description of Python's execution model is needed.

2.3 Python's Concurrent Execution Model

A concurrent execution model of a language consists of the units, the structure, and the semantics of concurrency. The concurrent units are the smallest, indivisible units of execution. The structure describes the coarse-grained organisation of these units and defines a (partial) order as well as ways to interact for these units. The concurrency semantics finally defines the possible outcomes of a concurrent program that follows the model.

Python instructions (`pyops`) are the units of Python's execution model. The units are structured in threads that execute with access to a shared memory. A thread is a stream of `pyops`, defining a partial order for these units based on the in-thread sequential order (program order). Multiple threads can run concurrently and interact with each other through shared memory. Each `pyop` can freely access and modify the shared memory, and, importantly, executes atomically.

`pyop` atomicity originates from a choice to use a *Global Interpreter Lock* (GIL) in Python's reference implementation *CPython* [8], which suffers from the same lack of parallel execution as the PyPy VM. Whenever CPython executes a `pyop`, the GIL must be exclusively acquired during the whole execution of that `pyop`. Hence, each `pyop` executes atomically without overlapping with the execution of `pyops` from concurrent threads.

2.4 Implementing Python's Execution Model

Because interpreters written in RPy are executable, they follow the execution model of RPy. Thus, to implement Duhton's or Python's execution model, the respective interpreters need to emulate these models within RPy's model. However, because RPy is a subset of Python, it also inherits its execution model. Hence, interpreters, like Python programs, are provided with instruction-level atomicity and suffer from a lack of parallel execution.

Within the PyPy interpreter, RPy functions implement the semantics of each `pyop`. For example, there is a `pyop` for multiplying two arguments together, and there is a corresponding RPy function that implements the semantics of that `pyop`. Hence, for the execution of a single `pyop`, several lines of RPy code are executed. To support `pyop` atomicity,

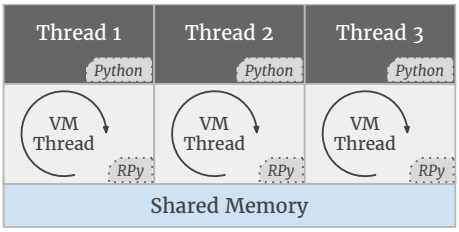

Figure 2. The PyPy VM's architecture for executing multi-threaded Python programs with VM threads.

Listing 3 Pseudocode of PyPy's interpreter loop

```
while True:
    instr = fetch_pyop()
    execute_pyop(instr)
    break()
```

the PyPy interpreter thus needs to ensure atomic execution not just for a single RPy instruction, but for multiple lines and functions. Hence, even if RPy's execution model provides instruction-level atomicity, the interpreter requires another mechanism to ensure atomicity for several instructions together.

To emulate Python's execution model, PyPy uses the architecture shown in Figure 2. Each Python thread executes within a *VM thread* that executes the core interpreter loop of PyPy's interpreter (Listing 3). Each VM thread can access the shared memory without restrictions. However, the loops within the threads do not execute in parallel; instead, only one loop progresses at any time and a `break()` instruction within the loop periodically yields the execution to another VM thread.

With this design, the PyPy interpreter ensures no interference by other threads for lines 1 to 3 of Listing 3, i.e., atomic execution for whole `pyops`. Interference from other threads, and thus a break of atomicity, only happens with the execution of the `break()` instruction, which is placed in-between the execution of two consecutive `pyops`.

Unfortunately, the only implementation of `break()` available with the current RPy framework is that of releasing and acquiring a single, global lock, i.e., a GIL. Further, the GIL is required to protect RPy internals for all interpreters that implement concurrency with VM threads. Hence, the GIL is a limitation of the RPy framework itself, affecting all such interpreters written in RPy.

To lift this limitation, we introduce the concept of quantised atomicity in RPy's execution model and thereby detach the model from any specific implementation of concurrency control. With quantised atomicity, we separate the means of specifying concurrency semantics with RPy from the implementation of the model by providing a native way for emulating atomicity guarantees of interpreted languages.

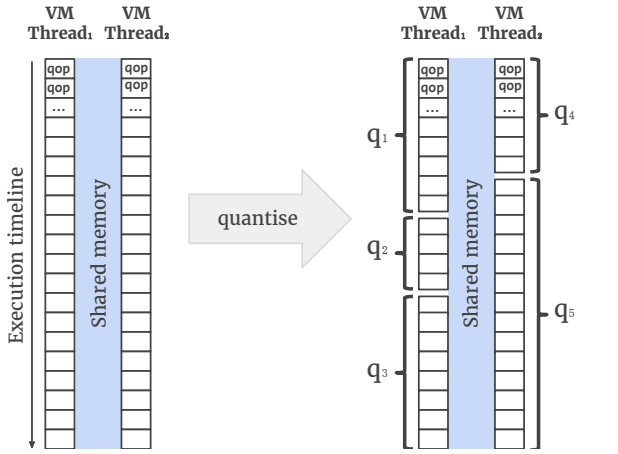

Figure 3. The concurrency model of Parallel RPy, which combines multi-threading in shared memory with quantised atomicity.

3. Parallel RPython

In this section, we introduce quantised atomicity in RPy's execution model. Quantised atomicity offers a new way to specify concurrency semantics in an interpreter and enables parallel execution. We refer to the new, extended RPy as *Parallel RPy*.

3.1 Parallel RPython's Execution Model

Interpreters written in RPy get translated into graphs of indivisible operations, which we call `qops`. `qops` are to RPy programs what `pyops` are to Python programs, but these `qops` follow Parallel RPy's execution model. Parallel RPy's model (Figure 3) inherits a lot from Python's: It, too, is a threading model with shared memory. VM threads define the structure of concurrency, each executing a sequence of operations (`qops`). `qops` are the concurrent units that either perform a calculation in local registers or access the shared memory. Multiple threads run concurrently, but in contrast to Python's `pyop` atomicity, Parallel RPy has *quantised atomicity*.

Quantised atomicity [6] is a form of atomicity that divides the sequence of `qops` in a thread into quantum regions (*quanta* $\mathbf{q_i}$ in Figure 3), each of them guaranteed to execute atomically and partially ordered by their in-thread sequential order (program order). A concurrent execution of `qops` must have the same outcome as some serial execution of the quanta that contain the `qops`, i.e., each quantum schedule must be serialisable. One valid concurrent execution of the quanta from Figure 3 is depicted in Figure 4.

Each quantum can contain an arbitrary number of `qops`. Quanta are delimited with `qbreak()` instructions, i.e., "quantum breaks", which the writer of the interpreter inserts. `qops` before a `qbreak()` belong to a different quantum than `qops` after the instruction. Or in other words, a `qbreak()` instruction ends the current quantum and begins

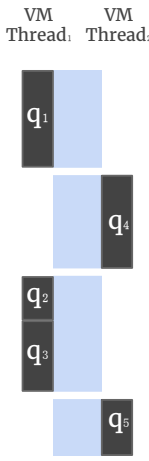

Figure 4. A conceptual view of concurrent execution of quanta on two VM threads.

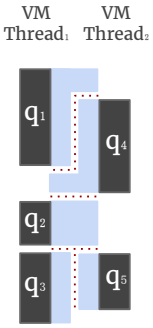

Figure 5. Overlapping, parallel execution of quanta by synchronising accesses to shared memory.

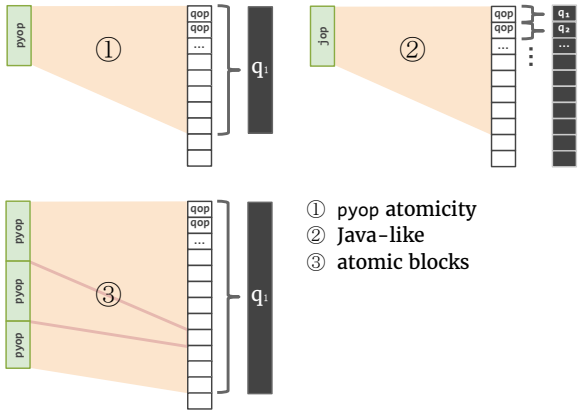

Figure 6. Three examples of mappings from interpreted instructions over `qops` to quanta.

a new quantum that contains all `qops` executed until the next `qbreak()` instruction.

Quantum breaks give the interpreter writer the ability to define consciously what must be executed as an atomic unit and where atomicity can be broken. Importantly, the division into quanta is not static, but can depend on runtime information. I.e., an interpreter can break atomicity conditionally based on the interpreted program. Such dynamic quanta allow mapping the different atomicity guarantees of interpreted languages onto Parallel RPy's execution model as we will discuss in the following section.

3.2 Implementing Execution Models with Quanta

With the introduction of quantised atomicity, an interpreter can emulate a language's execution model by mapping atomic units of that language to RPy's quanta. In PyPy's case, the `break()` instruction of Listing 3 can be seen as a marker that ends the current quantum and begins a new one, i.e., a quantum break. In that location, the break defines a quantum to contain the remainder of the loop body and thereby guarantees atomic execution for lines 1 to 3, i.e., the execution of a `pyop`. Essentially, there is a one-to-one mapping from `pyops` to quanta. This mapping is shown as example ① in Figure 6. The interpretation of a `pyop` involves the execution of several `qops` from the interpreter. To achieve `pyop` atomicity, these `qops` all map to a single quantum.

Other languages may require different mappings: A language similar to Java, whose programs may consist of `jops` (Java operations) that do not require atomic execution, can give each `qop` involved in the interpretation of a `jop` its own quantum (example ②).[1] Such a fine-grained mapping gives an implementation of Parallel RPy's execution model more opportunities to break atomicity, which may result in better performance.

On the other side of the spectrum of quantum mapping granularity is, e.g., support for atomic blocks in a language like Python (example ③). Atomic blocks enforce the atomic execution of all contained `pyops`. To achieve atomicity for a group of `pyops`, the `qbreak()` instruction in the interpreter loop executes conditionally, i.e., only if the current `pyop` is not within an atomic block. Thus, several `pyops` and their corresponding `qops` are mapped to a single quantum, and thereby are guaranteed atomic execution through quantum atomicity.

Defining concurrency semantics by mapping onto quanta is a natural approach. However, for some languages that do not require atomic execution at all, not even for single `qops`, this approach gives guarantees that are too strong. As a result, performance may be impaired. Finding ways to weaken atomicity for specific cases and in safe ways may improve the situation for such languages, but exploring such directions is outside the scope of this paper.

Importantly, interpreters do not deal directly with the actual implementation of RPy's execution model. Instead, they utilise the abstraction of quanta to implement an execution model and specify its concurrency semantics. Quanta make correct synchronisation a concern of the RPy framework where previously the interpreter had to synchronise the execution itself. As a result, we achieve the necessary separation that enables the development of alternatives to the GIL as a concurrency control component within the Parallel RPy framework.

3.3 Parallel Execution for Quanta

`break()` from the current RPy framework supports atomic execution for quanta by protecting them with a single global lock, a GIL. The GIL serialises the execution of quanta on multiple VM threads as depicted in Figure 4. However,

[1] An additional transformation in the RPy toolchain can insert the `qbreak()` instructions between `qops` automatically.

in some cases, quanta from different threads could run in parallel without violating quantum atomicity, as illustrated in Figure 5. If, e.g., the `qops` of a quantum q_3 all access different memory locations than the `qops` of a quantum q_5, i.e., q_3 and q_5 cannot influence each other, then executing q_3 and q_5 in any order produces the same outcome. Even a parallel execution of q_3 and q_5 has the same outcome and is therefore a serialisable quantum schedule. Other cases, represented by q_1 and q_4 in Figure 5, may still allow a partly overlapping execution of two quanta from different threads.

Such (partial) parallel execution is achievable for *statically* defined quanta using fine-grained locking. With this method, projects like Jython [2] and JRuby [9] avoid the GIL for the languages Python and Ruby. However, using fine-grained locking to implement Parallel RPy's execution model based on *dynamic* quanta proves to be prohibitively difficult: To provide atomicity for quanta, fine-grained locking needs to conservatively lock all resources required by a quantum before its execution. However, if quanta are defined dynamically, no static analysis of the interpreter can accurately infer the required resources in general. Thus, fine-grained locking is not a viable alternative to the GIL within the RPy framework. Instead, a dynamic approach to synchronisation is needed.

Instead of having to exhaustively and conservatively collect all resources that a dynamic quantum may need before its execution, a dynamic approach can make use of actual runtime information and thereby find the exact set of resources. In addition, a dynamic approach can be optimistic and can correct overly optimistic decisions at runtime. For these reasons, a dynamic synchronisation approach, such as one based on transactional memory, is currently the most appropriate choice for the Parallel RPy framework.

3.4 Quantum Atomicity with Transactional Memory

Transactional Memory (TM) provides the abstraction of *transactions* for synchronising concurrent accesses to shared memory. A transaction allows a group of operations to execute in complete isolation from other groups, and to make each group's execution appear to happen atomically. TM further guarantees that all possible transaction schedules are serialisable, meaning the outcome of a schedule is equal to the outcome of some serial execution of its transactions.

TM also supports optimistic concurrency control: Assuming that two concurrent transactions rarely access the same memory location, transactions are started in parallel. If, however, two transactions access the same location and at least one of the accesses is a write, the two transactions have a *conflict*. In that case, one transaction may be rolled back (or *aborted*) to restore atomicity and isolation guarantees, since not both transactions can *commit* their state to the shared memory without violating transaction schedule serialisability.

Supporting Parallel RPy's execution model with TM is straightforward: Because quanta provide roughly the same guarantees as transactions, we map one (or multiple) quanta to a transaction. Thereby, quantum atomicity and quantum schedule serialisability is guaranteed. Further, at runtime, the TM component may attempt to start multiple transactions in parallel on multiple threads. As a result, the TM component can reduce the overall runtime of multi-threaded programs running on multi-threaded VMs by overlapping the execution of quanta in time, similar to the illustration in Figure 5.

There have been previous attempts [3; 4; 5] of replacing the GIL with *Hardware TM* (HTM), i.e., TM implemented in hardware. HTM promises good performance and works transparently. However, hardware has inherently stricter limits than software. Current implementations of HTM limit the size of dynamic quanta, thereby hurting backwards-compatibility of the RPy framework. By exposing such an implementation limitation, the goal of separating specification and implementation aspects is violated. Hence, we chose *Software TM* (STM) to avoid this restriction. But we do not dismiss the possibility that future HTM generations, or even hybrid solutions, may lift that limitation.

4. Introducing Software Transactional Memory to the RPython Framework

In this section, we introduce the Software Transactional Memory (STM) component to the RPy toolchain. With that, the RPy toolchain gains the ability to switch between a concurrent GIL component and a parallel STM component for Parallel RPy's execution model.

4.1 Executing Quanta Atomically

The RPy toolchain's translation performs several transformations on the interpreter it receives as its input. Some of the transformations are standard compiler optimisations, such as inlining and constant folding, but others mix in Virtual Machine (VM) components that are controlled by hints within the interpreter. The transformation that adds the concurrency control component for Parallel RPy's execution model is an instance of the latter. With hints that delimit the quanta, that transformation ensures quantum atomicity in concurrent as well as parallel executions on multiple VM threads.

The `qbreak()` instruction is a hint that ends the current dynamic quantum and begins a new one. These `qbreak()` instructions mark the points within the interpreter where breaking quantum atomicity is allowed. Listing 4 revisits the example of PyPy's interpreter loop from Listing 3 and illustrates that the version using quanta to implement `pyop` atomicity looks essentially the same.

Listing 4 The PyPy interpreter's simplified dispatch loop expressed with quanta in Parallel RPy

```
while True:
    instr = fetch_pyop()
    execute_pyop(instr)
    qbreak()
```

The execution model transformation can work in two ways, depending on which component is chosen as its implementation. If the choice is made to use the GIL component, the `qbreak()` instruction translates to the original `break()` function that uses an explicit lock. But if the choice is made to use the STM component, the instruction translates to a function that may commit the current transaction and start a new one; thus, mapping quanta to transactions.

Since `qbreak()` is a hint and therefore only a *potential* point of breaking atomicity, both, the GIL as well as the STM component, can ignore one or several `qbreak()` instructions with the effect of merging multiple quanta together. For both components, merging has the advantage that the cost of a real `qbreak()` can be amortised by making quanta longer. Releasing and reacquiring the GIL, or committing and starting a new transaction in-between every quantum can be too expensive for fine-grained quantisation.

With the GIL component, a series of `qops` executed between two `qbreak()` instructions, i.e., a quantum, executes sequentially while no other thread can progress. The GIL achieves quantum atomicity by disallowing all concurrent execution. But the STM component must ensure a quantum's atomicity even if another thread progresses in parallel. Hence, quanta need to be isolated from other quanta. To enable this isolation, the transformation inserts read and write barriers before `qops` to let the STM component manage all accesses to shared memory that could otherwise break the isolation.

With these barriers, the STM component can create different views on shared memory. Each transaction can see a different version of an object, thereby isolating the transaction from others. The barriers further manage the read and write sets of transactions. These sets determine if a transaction is in conflict with committed state and, thus, needs to abort, or if its changes can be successfully committed.

Hence, the STM component can, with the help of barriers, isolate the quanta of Parallel RPy code. However, not every part of a VM produced with the RPy framework is written in RPy. Non-RPy parts are not under the control of the RPy toolchain, do not have the necessary barriers, and can therefore not be isolated easily with STM. Hence, we need to integrate such external code with the execution model of Parallel RPy and its quanta.

4.2 Invoking External Code

VMs commonly interface with external code in libraries and the operating system. Particularly for dynamic language VMs, performance critical functionality is frequently implemented externally in low-level languages that are more amenable to aggressive optimisations. Examples of such functionality are numerical algorithms, existing libraries, and system calls. We refer to such code as *VM-external*.

The invocation of VM-external code can be seen as a special type of `qop`. However, executed as part of a quantum, VM-external code may break the isolation and atomicity between quanta executing in parallel. Unfortunately, the RPy toolchain cannot know statically what RPy code may be part of a quantum, as quanta are delimited dynamically with the `qbreak()` instruction. Missing a way to prove the absence of external code invocations within a quantum, enforcing a restriction on the types of `qops` that are allowed within a quantum is thus impractical. Instead, all types need to be supported. Hence, we need to fit the invocation of external code into our execution model and explain how implementations can deal with it.

4.3 Categorising `qops`

Since not all VM-external code behaves the same and we model VM-external code as special `qops`, a categorisation of all `qops` according to their behaviour and possible effects on concurrent quanta is the first step. There are three basic categories:

I Fully managed `qops`
II External-atomic `qops`
III Arbitrary-effect `qops`

Category **I** are *VM-internal* `qops`: These `qops` are fully under the control of the RPy toolchain. In particular, they access only managed shared memory or operate only in local registers. On the other hand, Categories **II** and **III** are the *VM-external* `qops`.

In Category **II** are `qops` that invoke atomic, VM-external code. Such external code must guarantee that no VM-internal code or data is influenced by its execution and that it executes atomically as seen by other VM-external and -internal code. This type of atomicity is common in the exported functions of libraries with a thread-safe implementation.

Finally, Category **III** contains VM-external `qops` that represent VM-external code with unknown effects and without any useful atomicity guarantee[2]. An example is a call to a library that is not thread-safe, i.e., not safe to call from multiple threads in parallel.

4.4 Integrating VM-External `qops` with Quanta

Based on the possible effects of `qops` in each category, the `qops` can be integrated into the execution model of Parallel RPy in different ways.

Category **I** is already covered by the description of the execution model in Section 3.1. Quanta made up of only Category **I** `qops` are VM-internal quanta. These quanta are delimited explicitly by `qbreak()` instructions. In Figure 7, Scenario **A** shows the execution of two consecutive quanta, interrupted by a `qbreak()`.

A `qop` in Category **II**, however, can be mapped in two ways: First, it can execute in its own, VM-external quantum due to the strong atomicity guarantees of the invoked external code, which makes the invocation appear to happen atomically in the view of every other quantum. Second, it can be

[2] Also in this category are `qops` that access memory that is not managed by RPy (*raw* memory). For simplicity we consider these as VM-external.

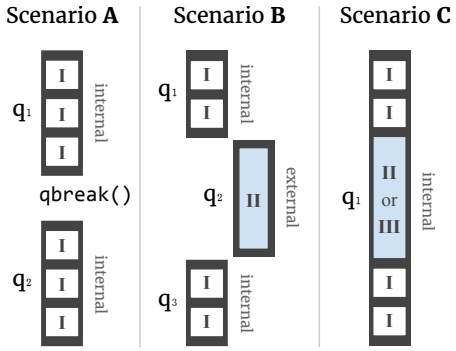

Figure 7. Three scenarios for integrating the categories of `qops` within Parallel RPy's execution model. Scenario **A** and Scenario **C** use only VM-internal quanta, but Scenario **B** also uses a VM-external quantum.

embedded in a VM-internal quantum, thereby inheriting the quantum atomicity guarantee. Both scenarios are shown in Figure 7 as Scenario **B** for the former, and Scenario **C** for the latter mapping.

Finally, `qops` in Category **III** *must* be embedded in a VM-internal quantum (Scenario **C**), because they do not provide any atomicity guarantees themselves. The enclosing VM-internal quantum must ensure atomicity and provide protection from any concurrent quantum.

4.5 Supporting VM-External Code with the GIL and the STM Component

The GIL component. Supporting the scenarios shown in Figure 7 is straightforward with a GIL: The execution of quanta is serialised and can, thus, protect even embedded external code easily (Scenario **C**).

Scenario **B** is exceptional since it enables a bit of parallel execution even with the GIL component. VM-external quanta guarantee their own protection and therefore do not need the GIL. Instead, the GIL stays available for any VM-internal quantum that is ready to execute. Consequently, a VM-internal quantum can execute in parallel to any number of VM-external quanta. Thus, in PyPy and other GIL-based VMs, Python code can run in parallel to external code in Category **II**. However, Scenario **B** is the only such exception; all VM-internal quanta are forced to run serially.

Category **II** `qops` can be handled by the GIL with either Scenario **B** or Scenario **C**. For long-running `qops`, Scenario **B** is preferable since it benefits from parallel execution. But for short-running `qops`, the overhead of breaking the VM-internal quantum cancels out the potential benefits, and thus, Scenario **C** is preferable.

The STM component. With STM, multiple VM-internal quanta can run in parallel if the underlying transactions do not conflict. STM can readily support Scenario **A**, which is required for `qops` in Category **I**.

Conveniently, the advantage of Scenario **B** also applies to STM: VM-external quanta are synchronised outside of STM, do not break the isolation of transactions, and therefore do not require the protection of a transaction. Hence, they can run in parallel to any transaction and, thus, to any VM-internal quantum.

Scenario **C**, however, cannot be directly supported by STM. VM-external `qops` can break the isolation of an enclosing transaction. These `qops` in general cannot be undone, and this property is required if a transaction needs to abort. Hence, even short-running Category **II** `qops` would require Scenario **B** and suffer from the overhead of breaking the VM-internal quantum. But fortunately, there is still an important subcategory of `qops` in Category **II** that never break the isolation of a transaction and never need to be undone: the subcategory of *pure* `qops`.

Purity for `qops` requires that the result of the computation depends only on the arguments of a `qop` and that there are no observable side-effects. These properties guarantee that the isolation between transactions cannot be broken and that aborting a transaction does not require undoing any effects of pure `qops`. An important group of `qops` with these properties are math functions, such as `sqrt()` or `log()`. These `qops` can still be handled with Scenario **C** by the STM component without additional overhead.

Unfortunately, `qops` in Category **III** are neither pure nor can they be handled by Scenario **B**; and thus, they absolutely require Scenario **C**. For these cases, STM allows transactions to turn *inevitable* [10; 11] directly before executing such `qops`. After a transaction turns inevitable, it is guaranteed to be able to commit. However, as a consequence, there can only ever be one inevitable transaction at any time. Hence, inevitable transactions are considered to be expensive. If many concurrent transactions need to turn inevitable, their execution is essentially serialised. In the worst case, transactions rarely execute in parallel and essentially revert to a serial execution similar to the GIL's.

With the description of how external code is integrated within Parallel RPy's execution model, and how the two components handle it, translating the same interpreter into a VM with either a GIL component or an STM component for controlling concurrency becomes possible. Both options are equal based on functionality. However, the two components behave differently when it comes to performance, which we discuss with the case study in the following section.

5. PyPy Interpreter Case Study

The PyPy interpreter is a Python interpreter written in RPy. It is in use by several companies to speed up their Python environment, which makes PyPy an interesting case for studying the challenges and issues that arise when porting a real-world interpreter to Parallel RPy and its STM component.

Ideally, switching over from `break()` to the `qbreak()` instruction is all that is required to produce a VM capable of executing Python threads in parallel. However, certain patterns of code are detrimental to the performance of STM as

they prevent successful speculative parallel execution. Below, we describe some of the issues we encountered within the PyPy interpreter.

5.1 The Performance Impact of Software Transactional Memory

The read and write barriers needed by STM add overhead to the execution of a program, but the amount depends on the actual STM implementation, which is not the focus of this paper. Universally harmful, however, are conflicts between transactions. With two transactions that have a conflict, only one can successfully commit; the other needs to abort and undo its changes, and the work done in parallel is lost. It is therefore essential that the number of conflicts stays low for parallelisation to pay off.

Importantly, there should be as few conflicts as possible caused by the VM itself. If, e.g., the interpreter causes a conflict whenever two concurrent transactions each call a method, it will be impossible for an application to avoid these conflicts and reach good performance. And indeed, with no additional modifications, the PyPy interpreter contained sources of such conflicts.

5.2 Sources of Conflicts within the Interpreter

Because performance is an important aspect for real-world interpreters, the PyPy interpreter also implements a number of techniques to increase interpreter performance[3]. One optimisation that proved to be problematic for the performance of an STM supported VM is *inline caching*.

Inline caching is a common technique in dynamic language interpreters for avoiding most of the overhead of repeatedly looking up a method or an attribute of an object. By maintaining a cache of the most recent lookup at each lookup site, the interpreter avoids repeatedly going through the inheritance hierarchy to figure out which method or attribute a name refers to.

The PyPy interpreter implements a variant of monomorphic inline caching that stores a table per code object (e.g., a Python function). The table has as many entries as there are names in that code object, and each entry stores the result of the most recent full lookup of that name. Thus, several lookup sites with the same name share an entry in the table.

Because inline caching is an interpreter feature and, e.g., depends on the interpreted language's name-lookup semantics, it is part of the interpreter and written in RPy. Hence, the table is a regular RPy object that is now managed by the STM component. It follows that updating the table from multiple threads can be a source of conflicts and cause transactions to abort. These conflicts are unnecessary, since the cache does not *need* to be updated. If the interpreter does not find the required entry in the table, it simply performs the full lookup. The full lookup never modifies objects and thus does not cause conflicts.

One way to avoid these conflicts is to make the caches thread-local. However, thread-local caches duplicate memory and need to be warmed up in each thread separately, which is not ideal. Furthermore, for checking the local cache, an additional indirection is required for retrieving the cache belonging to the current thread. For these reasons, we decided to attack the problem of caches more directly. However, our solution depends on the very STM system that we use, but similar solutions may work for other systems.

With support from the STM component, Parallel RPy provides a way of allocating an object that never causes conflicts even if modified concurrently. However, any modifications to that object may get lost. That semantics allows for an efficient implementation in our STM system and enables fast warm-up of inline caches with no additional indirection or memory duplication.

Other than the inline caching optimisation, there were a few more places where we had to make previously global data thread-local to avoid conflicts from concurrent modifications. With these changes, the PyPy interpreter ran without major sources of conflicts.

5.3 Performance Evaluation

To get a better idea of the performance of a PyPy VM with a GIL component versus a PyPy VM with an STM component, we compare the two on a set of benchmarks. The benchmarks are multi-threaded Python programs that distribute some workload on a set of threads. If more threads are available, less work must be done per thread.

On *PyPy-GIL*, the PyPy VM with the GIL component, we expect no change in the overall runtime of a benchmark when increasing the number of available threads, since the GIL serialises the execution over all threads and, thus, cannot utilise multiple cores of the CPU to perform work in parallel.[4] If there is any change at all, we expect the runtime to increase with more threads due to the additional overhead of managing threads and distributing the work.

On *PyPy-STM*, the PyPy VM with the STM component, we expect to see shorter runtimes with an increasing number of available threads. However, if just one thread is available, the runtime will be longer than the runtime of the same benchmark on PyPy-GIL. The worse runtime follows from the additional overhead caused by STM's read and write barriers, by additional synchronisation work when committing and starting transactions, and by object version management. We refer to these sources of overhead collectively as the *STM overhead*. With multiple threads, however, we expect the STM overhead to be amortised by the benefits of distributing the work on more threads and actually performing work in parallel on multiple CPU cores. However, the concrete advantage of PyPy-STM with multiple threads over PyPy-GIL

[3] PyPy also has a JIT and the performance of JIT compiled code is often independent of interpreter performance. However, some optimisations affect both, and both are important for whole-VM performance.

[4] The execution time of Category **II** `qops` is negligible in all benchmarks.

will depend on the concrete STM overhead observed in each benchmark.

Environment The evaluation was performed on a machine with 64GB of RAM and an Intel® Xeon® CPU E7-4830 with 4 NUMA nodes. Each NUMA node has 8 cores and benchmarks were pinned to one node to hide possible NUMA effects. The operating system was Ubuntu 12.04.5.

PyPy VMs The PyPy VMs were produced from exactly the same PyPy interpreter, once with the GIL component, once with the STM component. The translations were done without adding RPy's just-in-time compiler to either VM.[5] Besides the concurrency control component, the VMs also use different memory management components: STM needs its own, STM-aware garbage collector, which has a shorter history of tuning than the one used with the GIL component.

The final step from C to native code was done with a patched GCC 5.1.0 and the flags `-fno-ivopts`, `-fno-tree-vectorize`, and `-fno-tree-loop-distribute-patterns`. The patches were necessary for that version of GCC to compile PyPy-STM[6], and the flags disable some optimisations that produced incorrect results[7]. All code is available on Bitbucket [12] and further details about the STM system are described in a technical report [13].

Benchmarks Because threads in Python are currently only useful for their concurrency but not for performance, we did not find existing multi-threaded benchmarks. Hence, we collected a set of small-scale Python programs [14] that we then parallelised using threads and made into benchmarks. Here follows a short description of the benchmark set:

- `btree, skiplist`: These benchmarks insert, remove, and find elements in a data structure from multiple threads. Importantly, they make use of experimental atomic blocks that we introduced to Python. These blocks guarantee atomicity for a group of Python instructions (`pyops`) by ensuring that the block's execution happens fully within a single quantum (see ③ in Figure 6). Each of the aforementioned operations happens inside such an atomic block. On PyPy-GIL, the atomic block is a critical section implemented with a lock.
- `nqueens, mersenne, mandelbrot, perlin, raytrace, richards, parsible-bench, regex-dna, fannkuch-redux, k-nucleotide`: All of these benchmarks perform independent computations on multiple threads. Their concurrency generally follows the fork-join pattern.

[5] Results with the JIT are comparable, but explaining certain effects observed in the results require a deeper understanding, and thus, a discussion of the JIT internals, which we cannot provide in this paper.

[6] Some version of the patches is included in the newer GCC 6. We make ours available on request.

[7] On GCC 6, some of the optimisations seem to be fixed.

Measured times are reported as the average and the standard deviation computed from 30 iterations of a benchmark. These 30 iterations were collected in each configuration of each benchmark over 5 VM instances, each doing 6 in-VM iterations. There are 8 configurations resulting from 2 VMs, PyPy-GIL and PyPy-STM, and for each VM we set the number of available threads to 1, 2, 4, and 8.

Results Table 1 shows the results of this evaluation. As expected, PyPy-GIL does not benefit from more worker threads and the best runtime was always measured on 1 thread. The runtime generally worsens with every additional thread as the management overhead grows and contention on the GIL increases.

However, in the case of `btree` and `skiplist`, the overhead on multiple threads is unexpectedly high (233% and 137%). We suspect the cause to be the lock used to implement the critical section in these two benchmarks. On a single thread, that lock is always available. Lock implementations are optimised for such a case and, e.g., never do a system call into the kernel. On multiple threads, however, that lock is often acquired by another thread, causing the current one to call into the kernel to sleep. After releasing the lock, for another thread to wake up and acquire the lock is comparatively time-consuming. This hypothesis is supported by the observed number of system calls: When comparing runs on two threads versus runs on one thread, the number of system calls is around 600× greater[8]. The UNIX `time` command further reports that the two-thread runs do nothing for around 80% of the time.

The results for PyPy-GIL suggest that the use of threads usually makes a program slower in Python. This behaviour is not only counter-intuitive, but also makes threads a poor choice for concurrency use cases, such as performing a background task while keeping a responsive graphical user interface. With the approach described in this paper, threads may finally become a reasonable solution for concurrency in Python.

On PyPy-STM, the results mostly match our expectations. On just 1 thread, the benchmarks ran significantly slower than on PyPy-GIL (up to 74% on `k-nucleotide`). These results can be attributed to the STM overhead and PyPy-STM's less tuned garbage collector. From 1 thread to 8 threads, PyPy-STM scales well, achieving speedups between 3.77× and 7.59×. Except in the benchmarks `btree` and `skiplist`, where the speedups are only 2.17× and 2.14×.

The worse scaling of `btree` and `skiplist` results from our use of the experimental atomic blocks. Because some operations on the data structures conflict with others, using atomic blocks instead of a critical section means that transactions abort due to these conflicts. The other benchmarks, on the other hand, only rarely abort transactions, since they perform independent work in all threads. While for `btree`

[8] The increase comes exclusively from additional calls to the `futex` kernel API, which is used to implement the lock.

Python VM	PyPy-GIL				PyPy-STM				Max. speedup
Threads	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**	*
btree	**13.94** ±0.2	45.90 ±1.3	46.44 ±0.9	44.24 ±2.3	13.93 ±0.1	10.24 ±0.1	7.99 ±0.1	**6.42** ±0.1	2.17×
skiplist	**13.85** ±0.4	32.79 ±0.8	28.55 ±1.4	31.45 ±0.7	15.84 ±0.5	11.37 ±0.3	8.96 ±0.2	**7.39** ±0.2	1.87×
nqueens	**3.65** ±0.0	4.43 ±0.1	4.20 ±0.1	4.35 ±0.2	5.18 ±0.1	2.86 ±0.1	1.65 ±0.0	**1.30** ±0.0	2.81×
mersenne	**13.76** ±0.0	13.84 ±0.0	13.89 ±0.0	13.90 ±0.0	17.52 ±0.0	8.89 ±0.0	4.45 ±0.0	**2.31** ±0.1	5.96×
mandelbrot	**8.25** ±0.2	10.08 ±0.1	10.28 ±0.1	10.21 ±0.1	11.20 ±0.0	5.72 ±0.1	2.83 ±0.0	**1.48** ±0.0	5.59×
perlin	**28.48** ±0.5	34.97 ±0.7	35.22 ±0.3	35.08 ±0.3	37.75 ±0.2	19.53 ±0.1	**10.00** ±0.0	10.00 ±0.1	2.85×
raytrace	**17.71** ±0.1	19.85 ±0.1	20.38 ±0.1	20.31 ±0.2	21.66 ±0.4	11.36 ±0.2	5.87 ±0.1	**3.14** ±0.1	5.64×
richards	**11.68** ±0.1	14.01 ±0.5	13.77 ±0.1	14.00 ±0.1	16.90 ±0.1	8.92 ±0.0	4.67 ±0.0	**3.12** ±0.0	3.75×
parsible-bench	**5.67** ±0.0	5.97 ±0.0	6.07 ±0.0	6.17 ±0.0	7.19 ±0.0	3.99 ±0.1	2.33 ±0.1	**1.49** ±0.1	3.81×
regex-dna	**2.79** ±0.0	2.79 ±0.0	2.79 ±0.0	2.79 ±0.0	2.68 ±0.0	1.51 ±0.0	1.10 ±0.0	**0.71** ±0.0	3.90×
fannkuch-redux	**3.77** ±0.0	4.84 ±0.0	4.92 ±0.0	4.82 ±0.1	5.09 ±0.0	2.64 ±0.0	1.31 ±0.0	**0.68** ±0.0	5.55×
k-nucleotide	**12.10** ±0.1	15.79 ±0.2	15.19 ±0.1	15.40 ±0.1	21.06 ±0.1	11.71 ±0.1	7.21 ±0.1	**5.14** ±0.1	2.35×

Table 1. Maximum achievable speedup of PyPy-STM over PyPy-GIL on a set of benchmarks. The numbers represent the time in seconds of one benchmark iteration, reported as the mean and the standard deviation over 30 iterations. For each VM, the configuration with the best time is highlighted in **bold**. The last column shows the speedup achieved by the best PyPy-STM configuration over the best PyPy-GIL configuration.

and `skiplist` between 10% and 40% of transactions abort (increasing with the number of threads), in the other benchmarks that ratio stays below 5%. The comparatively high rate of conflicts in `btree` and `skiplist` thus limits their scalability.

Each benchmark benefits from parallel execution in PyPy-STM. Compared to the best result of PyPy-GIL, which is always on 1 thread, the best result of PyPy-STM, which is usually on 8 threads, is significantly faster. That maximum speedup over all configurations ranges from $1.87\times$ up to $5.96\times$. However, these numbers are the result of tuning some parameters of these benchmarks.

To reach these speedups, we had to make the amount of work for each benchmark reasonably high for the runtime to be longer than approximately 1 second. For shorter runtimes, PyPy-STM often does not yield any benefits. Further, we had to adapt the partitioning of the work into chunks so that first, there are enough chunks for 8 threads to be busy, and second, each chunk is large enough to amortise the overhead of dispatching it to a thread. Without adapting these parameters, PyPy-STM would often not improve the runtime of a program when compared to PyPy-GIL.

Furthermore, if there are sources of frequent conflicts within a program, the performance can quickly degrade. Hence, with the exception of `btree` and `skiplist`, our programs contain no systematic, avoidable sources of conflicts. Hence, we conclude that the approach can work well for multi-threaded programs that perform independent parallel work, but less so for irregular applications.

Understanding the performance of applications is further hindered by sources of conflicts being obscured by high-level operations in Python. Accessing an attribute, e.g., can invoke a method that contains a source of conflicts. Further, transactions abort because of one source, but when changing the program to remove that source, we sometimes find that there is another source directly following the removed one. The first source simply hid the one that came later. These obstacles make understanding an application's performance on PyPy-STM difficult.

6. Related Work

There have been several attempts [3; 4; 5] at replacing the GIL with TM directly in existing Python and Ruby VMs. These approaches use hardware TM (HTM) instead of software TM (STM), since HTM is supposed to work transparently with all native code. Because of this transparency, these approaches do not need to deal separately with VM-external code, since HTM handles both, VM-internal and -external code. However, especially the recent work [4], which exploits consumer-grade Intel® processors with HTM, shows the imposed restriction on the size of transactions, and also that current HTM often aborts transactions for other, hard to control reasons such as processor interrupts. For the Parallel RPy framework, the size limitation makes HTM already unsuitable, but the high rate of uncontrollable aborts suggests that using Intel's implementation of HTM may remain a challenge in the near future.

A project with a lot of similarities to the RPy framework is Truffle [15]. With Truffle, interpreters are partially evaluated to just-in-time compile the interpreted program to native code. In our context, an important difference to the RPy framework is that Truffle itself does not depend on a GIL. Instead, interpreters execute in Java's execution model. However, Truffle does not provide an abstraction comparable to quanta that allows interpreters to map atomicity requirements of the interpreted language onto Java's execution model. Hence, for implementing Python's execution model, for example, an explicit GIL, or manual fine-grained locking, is necessary.

Swaine et al. [16] introduce *slow-path barricading* for incremental parallelisation of VMs. They avoid a major

revision of the VM by distinguishing between operations that are safe to run in parallel (often the fast-path) and operations that are unsafe. Our approach avoids a major revision by working as a transformation of the RPy toolchain, making the whole VM safe for parallel execution, but requiring an RPy interpreter to begin with.

7. Conclusions

Since the single-core performance of processors is stagnating, Virtual Machines (VM) need to embrace parallelism to stay relevant. However, VMs developed with the RPython framework synchronise concurrent execution with the help of a GIL. Hence, parallel execution with threads becomes impossible.

By introducing a new execution model based on the concept of quantised atomicity to RPython, Parallel RPython obtains the powerful abstraction of quanta to emulate execution models of dynamic languages with RPython's model. As an alternative to the GIL, a Software Transactional Memory (STM) component supports the parallel execution of quanta with unchanged semantics.

In a case study, we evaluate the new execution model, once with a GIL component for concurrency control, and once with an STM component. PyPy, a Python interpreter, is modified to work with STM. While functionally equivalent, PyPy-STM needs some internal changes to reduce the rate of conflicts that otherwise cause unnecessary transaction aborts. With these modifications, PyPy-STM performs significantly better than PyPy-GIL on a set of multi-threaded Python programs if two or more threads are available. The achieved speedups range from $1.87\times$ up to $5.96\times$.

The results demonstrate that the transition to the new execution model to support parallel execution with STM can be straightforward and can yield significant speedups for programs that perform independent work on each thread. While the speedups are encouraging, achieving them is not always straightforward. Often, the performance of PyPy-STM is difficult to understand, since Python can obscure sources of conflicts, and sources can hide other sources. Therefore, it is essential that future work also considers performance debugging with an STM component as an important topic.

In summary, we bring the benefits of parallel execution to Python in a way that allows other interpreters using the RPython framework to reuse our work. Before becoming universally applicable, however, further research needs to explore ways to lower the overhead of STM when only one thread is active. Otherwise, the STM component cannot serve as a drop-in replacement for the GIL component in cases where a large percentage of the runtime is spent in non-parallelisable parts of a program.

Acknowledgements

We thank Michael Faes and Aristeidis Mastoras for all the helpful discussions and thank our reviewers for helpful comments, which pointed out many problems and greatly improved the paper.

References

[1] "The IronPython Project," `http://ironpython.net/`.

[2] "The Jython Project," `http://www.jython.org/`.

[3] Nicholas Riley and Craig Zilles, "Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution," OOPSLA '06, pp. 998–1008, ACM.

[4] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari, "Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory," PPoPP '14, pp. 131–142, ACM.

[5] Fuad Tabba, "Adding Concurrency in Python Using a Commercial Processor's Hardware Transactional Memory Support," *SIGARCH Comput. Archit. News*, vol. 38, no. 5, pp. 12–19, Apr. 2010.

[6] Yu David Liu, Xiaoqi Lu, and Scott F. Smith, "Coqa: Concurrent Objects with Quantized Atomicity," CC'08/ETAPS'08, pp. 260–275, Springer-Verlag.

[7] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo, "Tracing the Meta-level: PyPy's Tracing JIT Compiler," ICOOOLPS '09, pp. 18–25, ACM.

[8] "CPython," `https://www.python.org/`.

[9] "The JRuby Project," `http://jruby.org/`.

[10] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott, "Implementing and Exploiting Inevitability in Software Transactional Memory," ICPP '08, pp. 59–66, IEEE Computer Society.

[11] Colin Blundell, E. Lewis, and Milo Martin, "Unrestricted Transactional Memory: Supporting I/O and System Calls Within Transactions," *Technical Reports (CIS)*, May 2006.

[12] "PyPy Source on Bitbucket," `https://bitbucket.org/pypy/pypy`.

[13] Remigius Meier, Armin Rigo, and Thomas Gross, "A Transactional Memory System for Parallel Python," `https://bitbucket.org/pypy/extradoc/raw/12bda771698925b8296808959e7b830aef8b78b8/talk/dls2014/paper/paper.pdf`, Aug. 2014.

[14] "PyPy Benchmarks on Bitbucket," `https://bitbucket.org/pypy/benchmarks`.

[15] Christian Wimmer and Thomas Würthinger, "Truffle: A Self-optimizing Runtime System," SPLASH '12, pp. 13–14, ACM.

[16] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt, "Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems," OOPSLA '10, pp. 583–597, ACM.