Virtual Machine Design for Parallel Dynamic Programming Languages

REMIGIUS MEIER, ETH Zurich, Switzerland
ARMIN RIGO
THOMAS R. GROSS, ETH Zurich, Switzerland

To leverage the benefits of modern hardware, dynamic languages must support parallelism, and parallelism requires a virtual machine (VM) capable of parallel execution — a parallel VM. However, unrestricted concurrency and the dynamism of dynamic languages pose great challenges to the implementation of parallel VMs. In a dynamic language, a program changing itself is part of the language model. To help the VM, languages often choose memory models (MM) that weaken consistency guarantees. With lesser guarantees, local program state cannot be affected by *every* concurrent state change. And less interference allows a VM to make local assumptions about the program state which are not immediately violated. These local assumptions are essential for a VM's just-in-time compiler for delivering state-of-the-art VM performance.

Unfortunately, some dynamic languages employ MMs that give exceedingly strong consistency guarantees and thereby hinder the development of parallel VMs. Such is the case in particular for languages that depend on a global interpreter lock, which mandates a MM with sequential consistency and instruction atomicity.

In this paper, we reflect on a first implementation of the Parallel RPython execution model, which facilitates the development of parallel VMs by decoupling language semantics from the synchronization mechanism used within the VM. The implementation addresses the challenges imposed by strong MMs through strict isolation of concurrent computations. This isolation builds on transactional parallel worlds, which are implemented with a novel combination of software techniques and the capabilities of modern hardware.

We evaluate a set of parallel Python programs on a parallel VM that relies on Parallel RPython's implementation. Compared with a serial baseline VM that relies on a global interpreter lock, the parallel VM achieves speedups of up to 7.5× on 8 CPU cores. The evaluation shows that our realization of Parallel RPython meets the challenges of dynamic languages, and that it can serve as a solid foundation for the construction of parallel dynamic language VMs.

CCS Concepts: • **Software and its engineering → Runtime environments**; *Scripting languages*;

Additional Key Words and Phrases: dynamic language, parallel execution, virtual machine

ACM Reference Format:
Remigius Meier, Armin Rigo, and Thomas R. Gross. 2018. Virtual Machine Design for Parallel Dynamic Programming Languages. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 109 (November 2018), 25 pages. https://doi.org/10.1145/3276479

Authors' addresses: Remigius Meier, Department of Computer Science, ETH Zurich, Zürich, Switzerland, remi.meier@inf.ethz.ch; Armin Rigo, arigo@tunes.org; Thomas R. Gross, Department of Computer Science, ETH Zurich, Zürich, Switzerland, thomas.gross@inf.ethz.ch.

2475-1421/2018/11-ART109
https://doi.org/10.1145/3276479

1 INTRODUCTION

Virtual machines (VMs) that allow the execution of parallel shared-memory programs are difficult to develop. In particular, parallel VMs for popular dynamic languages such as Python and Ruby suffer from this problem. One reason is that these languages lack, or lacked for a long time, a memory model suited to parallel execution. Instead of introducing such a memory model when concurrent execution became a necessity, Python and Ruby introduced a single global lock within the VM to serialize the execution of multiple threads. This Global Interpreter Lock (GIL) prevents true parallel execution and, as a byproduct, gives strong guarantees, such as sequential consistency and atomicity for certain operations [Python Software Foundation 2018]. Unfortunately, because these guarantees are difficult to uphold in other ways, to this date, VMs for these languages often still depend on the GIL. Only a handful of truly parallel VMs exist, which all implement these guarantees through error-prone fine-grained locking [IronPython 2018; JRuby 2018; Jython 2018].

Parallel RPython [Meier et al. 2016] is a parallel execution model for dynamic language VMs. The model was presented in the context of the PyPy project [Rigo and Pedroni 2006] and builds upon this framework for constructing high-performance concurrent, but not parallel, VMs. Through quantized atomicity [Liu et al. 2008], the model provides a way to specify concurrency semantics in a parallel VM without the need to resort to fine-grained locking. Instead, the synchronization mechanism of the VM becomes independent from the concurrency semantics of the language.

Our earlier work [Meier et al. 2016] describes this execution model, its semantics, how concurrency semantics of dynamic languages can be expressed with its help, and how the model can be supported by transactional memory. In this paper, we present the realization of the model and the challenges that must be addressed by an implementation, and we describe the design of a transactional memory system for the model's synchronization.

The model's quantized atomicity guarantees that every computation is atomic by default for as long as there is no explicit atomicity breakpoint. Atomic computations can become arbitrarily long until such a breakpoint is reached, and after a breakpoint, the next atomic computation begins. These atomic computations are called *quanta*. The model requires that its synchronization mechanism guarantees atomicity for concurrent quanta without imposing constraints: Quanta have unrestricted access to all memory locations and they may execute in parallel. Additional restrictions for the design of a synchronization mechanism come from the functionality that dynamic languages offer to programmers. We present these requirements and constraints and show how they are addressed by the design of a Software Transactional Memory (STM) system.

We emphasize that the STM system is *not* exposed by the dynamic language as a synchronization mechanism for applications. Rather, the STM is the underlying implementation of quantized atomicity and serves as the foundation of a parallel VM. Serving as the backbone of the entire VM is a challenging requirement for the design of the STM. The parallel execution model underlies and protects all components of a VM, including the interpreter and the just-in-time compiler. Hence, all computations and all memory locations are protected by quantized atomicity, which means that the entire VM's memory is managed by the STM system.

Handling such a large memory space, not just isolated data structures, requires a particular design for the STM. We introduce the concept of parallel worlds which is enabled by a novel combination of software techniques and existing hardware capabilities. This combination is particularly synergetic and suited to meet the demands of dynamic languages. The resulting system provides — in a dynamic language VM — a platform for the parallel execution of a variety of application programs. The system extends PyPy's approach to VM construction from producing concurrent VMs to producing truly parallel VMs — VMs that are capable of running parallel programs and let them utilize the ever-increasing parallelism in modern hardware.

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

- A presentation of the unique challenges that dynamic languages pose to the design of parallel VMs, and in particular to a realization of Parallel RPython's execution model [Meier et al. 2016].
- A unique top-down approach to the design of an STM system, introducing the concept of parallel worlds.
- State-sharing across parallel worlds using the capabilities of a modern x86-64 CPU, such as virtual memory and memory segmentation.
- An on-demand, page-level memory sharing algorithm using the CPU's memory protection capabilities.
- An evaluation of a parallel Python VM that is built with Parallel RPython's execution model and the presented STM system.

2 PROBLEM STATEMENT

In this section, we present the main challenges that dynamic languages pose to the design of parallel virtual machines (VMs) in general, and to an implementation of Parallel RPython's execution model [Meier et al. 2016] in particular.

2.1 Dynamic Languages

Dynamic languages have a set of characteristics that makes many performance optimizations difficult [Chambers et al. 1989; Tratt 2009]. Here, our focus lies on the challenges posed to a dynamic language runtime system, i.e., to a dynamic language VM, if programs can execute code in parallel – one of the most important performance optimizations for modern hardware [Daloze et al. 2016, 2018].

The most visible characteristic of a dynamic language is its dynamism. Only few properties about a program written in a dynamic language are statically known before its execution, and during execution, properties can change unpredictably at any point in time. A VM for such a language needs to constantly adapt even to a program changing itself, which is one reason why dynamic languages are usually interpreted. An interpreter, as opposed to a compiler, rarely makes assumptions about a program and merely transforms the current program state according to the program's current instruction.

However, during the execution of many programs, many program properties remain constant for some time after an initial phase of adaptation. Such temporary phases of constancy can be exploited by a just-in-time compiler (JIT) to compile parts of a program that do not change for a while (or at least are expected not to change). These compiled parts can boost the interpreted program's performance close to the level of statically compiled programs. The JIT therefore is an essential component of any high-performance VM. Still, numerous assumptions that a JIT must make about the constancy of certain program properties can be violated by any part of the program. Hence, the compiled code needs to check the validity of its assumptions diligently. When an assumption is not valid anymore, execution may need to fall back to the interpreter again since compiled code cannot adapt easily to failed assumptions.

Correctly coping with the possibility of change puts the burden of change management on the JIT, which is one reason for making the JIT the most complex component of a dynamic language VM. JITs for the language JavaScript, as they can be found in all modern web browsers, have proven that high performance is achievable, despite the complexity. However, notably absent from JavaScript is a concurrency model where concurrent tasks can interact without restrictions through shared memory. A parallel programming model that is constrained in this way simplifies the job of a JIT considerably since assumptions can only be violated at explicit synchronization points in a program. Between synchronization points, a JIT can insert a check for an assumption in compiled code and let the following code depend on it. However, not all dynamic languages restrict their concurrency on the level of the language's programming model. With unrestricted concurrency, there may be no explicit synchronization points.

Defining memory models is another way for languages to restrict the possible outcomes of concurrent shared memory accesses. Memory models describe when and how memory modifications propagate and become visible to concurrent computations, thereby possibly weakening consistency guarantees. Such models can define synchronization points, in-between which a JIT can depend on certain assumptions staying valid. Thereby, JITs are partially relieved from the burden of change management and they can optimize the program assuming no interference from concurrent computations.

Restricting concurrency, or defining memory models that are suitable for parallel execution, is how many dynamic languages [Armstrong 2010; Kessler and Swanson 1990; Kraus and Kestler 2009; Mattarei et al. 2018] contain their dynamism and allow JITs to work effectively. However, a class of popular languages does not restrict concurrency and their designers also never consciously decided on a memory model suitable for parallelism: languages that depend on a Global Interpreter Lock (GIL).

For example, in the language Python, concurrently running code has unrestricted access to shared memory and can invalidate an assumption with every instruction; there is no guarantee that an assumption holds for any part of compiled code. This problem is worked around by depending on the global critical section provided by the VM-wide GIL. Within a critical section, assumptions cannot be invalidated by other code since no other code can execute concurrently. However, the critical section serializes the execution and prevents parallelism.

A GIL is hence useful in two ways: It provides a simple way to synchronize the VM under concurrent execution, and it enables a JIT to work effectively by providing critical sections. However, besides the evident disadvantage of preventing parallel execution through serialization, the GIL introduced another problem: its own memory model, which we will call the GIL Memory Model (GMM). Because of the serialization of threads, the GMM guarantees *unconditional* sequential consistency. Sequential consistency mandates that modifications propagate instantly to concurrent computations. Additionally, the GIL made it easy for VMs to make any operation happen atomically. In the case of Python, this lead to the explicit support [Python Software Foundation 2018] of atomic operations on built-in data types. Both, the sequential consistency and the support of atomicity, are exceedingly strong guarantees of the GMM, and they are difficult to uphold in a parallel VM.

Over time, Python and others quasi-standardized on the GMM because of Python's widely used standard implementation using a GIL. Hence, a compatible VM for Python is now required to implement the GMM also in a parallel VM.

> **Challenge 1** True and unrestricted parallel execution combines the complexity of concurrency and the dynamism of dynamic languages. Program properties can be changed by concurrently running code, which means that assumptions made by a JIT are not guaranteed to hold for any larger part of compiled code. Additionally, any solution must be able to provide sequential consistency and atomicity for supporting languages with a GMM.

Another, possibly surprising challenge of dynamic languages arises from their use for scripting. The utility for scripting demands from a language embeddability, and thus a simple way to interface with native code. Such interfacing capabilities require from otherwise fairly high-level languages a way to break all abstractions and deal directly with low-level, sometimes foreign data. Python, for example, is often used for operating system scripting, which requires giving programs direct access to operating system functions. With this level of access, programs can break through the transparency of Python's thread abstraction and inspect their own process or the host system.

One consequence is that Python's threads must be native kernel threads, they cannot be virtualized as green threads (user-level threads). Python programs could easily distinguish the two threading implementations by calling operating system functions. This power given to programs in dynamic languages restricts what can be *virtual* in virtual machines. Even though dynamic languages are high-level languages, VMs must still grant them low-level access and must not break their expectations.

> **Challenge 2** Since dynamic languages allow programs to break through abstractions, the runtime system must retain the expected thread and process encapsulations. A VM cannot virtualize everything, some integration with the host environment is imposed by the languages.

Our approach to implement a VM should be language agnostic at least to the point that different dynamic languages can be supported. Therefore, we cannot put arbitrary restrictions in place, and these two challenges make a parallel VM design particularly demanding. Hence, they serve as major drivers and constrain the design space for a parallel implementation of Parallel RPython's execution model.

2.2 Parallel RPython

RPython is a framework for developing VMs for dynamic programming languages. Given a language interpreter as the specification of the language semantics, RPython generates a complete VM with a garbage collector and a JIT [Bolz et al. 2009]. The Parallel RPython execution model was conceived as a way for language interpreters to additionally specify *concurrency semantics* in the RPython framework. The following paragraphs summarize the model [Meier et al. 2016].

The model allows specifying concurrency semantics with an abstraction called a *quantum*. A quantum is a sequence of instructions that is guaranteed to execute atomically. While similar to atomic blocks, there are essential differences: First, quanta are defined dynamically at runtime. Quanta do not enclose and protect a piece of code, but rather enclose and protect an executing *stream of instructions*. Second, quanta fully partition the stream of instructions. No instruction is executed outside of a quantum, and as soon as a quantum ends, a new one begins. This form of atomicity is called *quantized atomicity* [Liu et al. 2008].

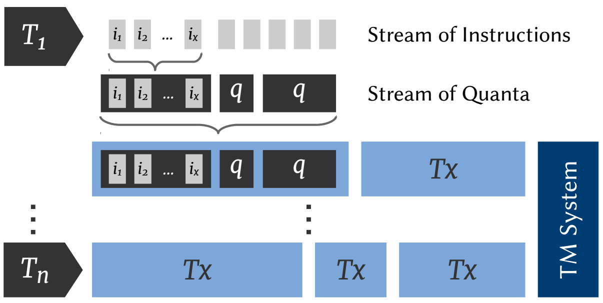

Fig. 1. Threads T_n producing streams of instructions i_x that are enclosed in quanta q. The streams of quanta are then enclosed in transactions T_x and executed by the TM system.

Figure 1 illustrates Parallel RPython's threading model: An arbitrary number of threads can run concurrently with each other, communicating through shared memory. Each thread produces a stream of instructions that is partitioned into quanta. Quantized atomicity guarantees that all quanta from all threads execute in such a way that even a parallel schedule of quanta is serializable, i.e., the same outcome is obtainable with a purely serial execution of all quanta. Hence, quanta allow implementing arbitrary atomic instructions in parallel VMs. Quantized atomicity therefore allows implementing the GIL Memory Model, addressing one part of **Challenge 1** presented in the previous section.

Ensuring serializability for quantized atomicity is the responsibility of a VM's synchronization component, for which we use a transactional memory system. Transactional memory (TM) is a concurrency control mechanism that supports enclosing a group of operations and their effects in transactions. The TM system then makes the effects of a transaction happen atomically. Most importantly, transactions can also speculatively run in parallel. The TM system then ensures atomicity by letting transactions commit their changes only if the outcome is serializable. If the transaction cannot commit, all its effects are undone and the transaction restarts.

The atomicity and speculative execution of transactions can be used to implement the guarantees of quanta while also supporting parallel execution. Figure 1 shows how quanta are enclosed in transactions and executed in parallel by the TM system. Thereby, dynamic language programs that produce multiple streams of quanta can execute these streams in parallel and achieve speedups on parallel hardware.

In RPython VMs, the language interpreter and also the JIT are protected by the quantum abstraction. Therefore, with few exceptions, everything in the VM runs in transactional memory. One important takeaway is that the TM system does not merely deal with a few data structures that are synchronized with transactions, but synchronizes all memory and all computations in the VM. Hence, the TM system plays a critical role as the backbone of the VM.

3 DESIGN OF THE SOFTWARE TRANSACTIONAL MEMORY SYSTEM

In the following sections, we first introduce the basic mechanisms of the software transactional memory system (STM). These mechanisms ensure atomicity for transactions, and consequently for quanta. Afterwards, we highlight the key design decisions that make the STM uniquely suitable for use as the backbone of a dynamic language VM.

3.1 Terminology

We first restate the transactional memory terminology used throughout this paper. Figure 2 shows the events a transaction encounters during its lifetime. Each transaction's life begins with a *start* and ends with either a successful *commit* or an *abort*. The commit is the atomic point where the effects of a transaction are applied to the globally visible state. This global state can only be modified by commits and is thereby always consistent (a transaction either "happened" or it did not). During the lifetime of a transaction, the transaction-local state may be modified, but it is not globally visible.

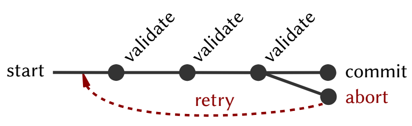

Fig. 2. Transaction events

An *abort* happens if a *validation* fails. The validation's responsibility is to check if the transaction's state is still consistent with the globally visible, committed state, i.e., after a transaction's *start* there was no *commit* of another transaction that made conflicting modifications. Modifications are conflicting if they invalidate a transaction's assumptions of the global state. For example, if transaction A reads the value of a variable x right after its *start*, later computations in A may depend on that value. If a transaction B changes variable x to hold another value and manages to successfully commit that change before A commits, then *validation* of A must fail since A's computations are based on an outdated value of x and A, thus, cannot commit its changes to global state without violating serializability. Hence, A *aborts*, which entails undoing all of its local modifications and restarting the transaction from where it originally *started*.

To support these events, the mechanisms of validating state, undoing changes, and atomically committing changes to the global state are required. In the following, we describe the high-level design of these mechanisms.

3.2 Validation

The STM algorithm works with object granularity. Validation, for example, detects conflicts on a per-object level, not on a per-field or word level. A conflict is detected if the first access to an object is earlier than a modification recorded in a recent commit. In that case, whatever the transaction saw is now outdated and it cannot successfully commit anymore.

Hence, the transaction needs to keep track of objects that it has read from or has written to. This recording of objects is done by read and write barriers. Before every read, the read barrier marks an object as read. Before every write, the write barrier, on first access, records the object in a set. That set is also part of what will be appended to a global *commit log* during a commit. A validation can then compare the set of written objects with new entries appended to the commit log since the last validation. And entries in the commit log can be checked if they mention objects that the transaction has marked as read. These checks are sufficient to determine if a transaction's state is consistent and may be committed. If these checks fail, the transaction must abort.

3.3 Aborting

Aborting a transaction requires undoing its effects, i.e., modifications to objects that happened during the transaction's lifetime. For that purpose, the write barrier makes a backup copy of each object on the first write within a transaction. During an abort, these backup copies are copied over the modified objects, thereby resetting all objects to the state they were in when the transaction started. This state is again consistent, albeit outdated.

Validation and aborting abilities are necessary for providing consistent states to transactions. But for allowing multiple parallel transactions to further the global state, the STM needs a way to commit changes to the global state. These commits must happen *atomically* to ensure serializability, and the changes must become visible in new transactions and existing transactions through validation.

3.4 Atomic Commit and Parallel Worlds

The STM isolates transactions from each other such that no transaction can observe the intermediate state of another transaction running in parallel. With this level of isolation, the STM creates the illusion of no concurrency, and transactions can access and modify objects without restrictions. Essentially, each transaction lives in its own *world*. Each world contains a consistent copy of the whole global program state, and modifications of a copy do not affect the copies of any parallel worlds.

To further the global state, modifications in one world must eventually become visible to all worlds. This *reconciliation* of worlds happens atomically in each world, but asynchronously to other worlds. I.e., reconciliation happens independently in each world as part of validation. With the help of the commit log, validations check for conflicts, but also import and apply the changes from commits that happened since the previous validation. Validations are done at the start of a transaction to get an up-to-date consistent world, and right before each atomic commit.

A commit is the linearization point of a transaction. Atomically committing works by repeatedly validating and attempting to attach a new entry to the global commit log. If multiple transactions attempt to commit at the same time, either validation or attaching a new entry may fail. In the latter case, the commit is simply retried; but in the former case, the only option is to abort the transaction since no future validation can succeed anymore.

3.5 Key Design Decisions

The STM builds on a model where each transaction lives its own world, which is completely isolated from other parallel worlds. Worlds are synchronized through a central commit log that keeps track of changes to the global state. Hence, to modify the global state, a transaction must commit changes by appending them atomically to the commit log. And to see those changes in parallel worlds, these worlds must validate and import any new entries from the commit log.

Complete world isolation is a deliberate design choice made to address **Challenge 1** described in Section 2.1. Without this degree of isolation, a JIT and the code it generates would have to handle any change that could happen in parallel. Such interference prevents many important optimizations and requires handling problems such as out-of-thin-air values. But with isolation, a JIT can depend on the STM system to prohibit any interference from parallel transactions. Hence, assumptions about program properties cannot suddenly become invalid, only at transaction boundaries (or more precisely, at quantum boundaries that are marked by explicit atomicity breakpoints).

However, the high-level description of the STM system does not explain all important issues. One such issue is that the isolation of parallel worlds cannot be done naively. Each world contains the whole state of the program, or more precisely, the state of the entire VM. If we were to naively create W worlds by making W copies of that state, the memory requirements would be prohibitive. In the following section, we thus introduce our approach for creating parallel worlds that do not suffer from such excessive memory requirements.

4 EFFICIENT PARALLEL WORLDS

We use the term *world* to describe a copy of the whole program state that is exclusively available to a transaction. This description is not merely an illustration for the complete world isolation of the STM; our approach takes the description literally and makes it efficient.

STM implementations often start with the assumption that there is a single program state (or even just a single data structure that is accessed by transactions), and only those accesses need to be synchronized. In this work, however, we start with the assumption that there are W program states. Each state is directly and fully accessible by a single transaction at a time, and the states must be reconciled continuously. This top-down approach leads us to a unique STM system design.

We start with the important insight that frequent reconciliation of worlds makes these worlds largely identical. Only local modifications and not-yet imported changes make worlds different. Hence, it is reasonable to share program state that is identical across all parallel worlds. In the following sections, we describe the difficulties attached to this idea and an approach that overcomes them.

4.1 State-Sharing Worlds

A discussion on the level of program state is too abstract to find a concrete solution for state-sharing between worlds. Essentially, at the implementation level, program state is (program) memory. And if identical program state should be shared, the underlying memory needs to be identical too.

With an object-based program state, program state not only consists of data values, but also of the references between objects. These references are addresses within an object's memory that point to the memory of the referenced object. Therefore, if two references in two different worlds refer to the same object, they *must* be identical addresses. Even if the two worlds reside at different locations in memory, references to the same object must be identical in memory, i.e., identical addresses. Hence, each object must have a single address that uniquely identifies it across all worlds.

To illustrate why this is a problem, consider the following example: An object o_A references object o_B in one of its fields. Hence, the memory of o_A somewhere contains the address of o_B. Now,

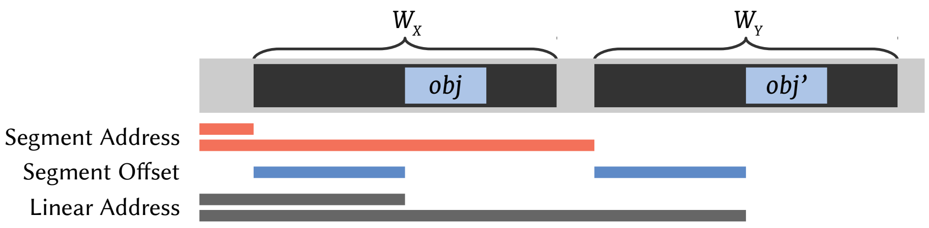

Fig. 3. Two segments W_X and W_Y in the virtual address space. The object *obj* is at the same segment offset in both, but at different linear addresses.

if in world W_X object o_B is modified by a transaction, that modification should not be visible in another world W_Z, i.e., there must now exist two versions of o_B. But if we want to share the memory of o_A between W_X and W_Z, the address of o_B must be the same in W_X and W_Z. That address must therefore resolve to different versions of o_B in different worlds, one version for W_X and one for W_Z.

Resolving an address differently in each world can be expensive. A simple implementation would require some kind of lookup for the right object version whenever a field is dereferenced. But the runtime overhead of such a solution is significant. However, we found that modern x86-64 CPUs offer a low-overhead solution to this problem based on the general mechanism for checking and translating addresses.

x86-64 CPUs support the *virtual memory* management technique, which offers each process its own 48 bits of address space. Through address translation, the **same** virtual address can map to **different** (physical) memory addresses in different address spaces, which is what a solution to the above problem must achieve. However, user space processes are prevented from creating new address spaces; only by using multiple processes can we create multiple address spaces.

Unfortunately, moving threads each into their own process to create separate worlds is not a solution. **Challenge 2** requires that a solution does not change the thread and process encapsulations. For some dynamic languages, threads cannot be virtualized and must all be part of the same process, otherwise applications that expect so could stop working.

While virtual memory cannot be used to resolve addresses differently in different worlds, it still offers a large virtual address space and, interestingly, the opposite of what we need: User space processes are allowed to control the address translation in their address space such that **different** virtual addresses map to the **same** physical address. This capability will be useful later for sharing memory between worlds.

The key to a solution is that x86-64 CPUs combine virtual memory with a lesser-known memory management technique: *memory segmentation*. Memory segmentation, as illustrated in Figure 3, allows for the creation of memory segments anywhere in virtual memory. Within these memory segments, *segment offsets* serve as segment-local addresses. Any x86 instruction that accesses memory can be prefixed such that the instruction works with a segment offset instead of a virtual address, and a special *segment register* selects the current segment. Hence, a **single** segment offset can resolve to **different** locations in memory, depending on the selected memory segment.

Combined with virtual memory, the entire hardware supported address translation from a segment offset to a physical memory address happens in these two steps:

(1) The *segment offset* is added to the segment address that is stored in the *segment register*, which yields a *linear address* pointing to a location in the virtual address space (Figure 3).
(2) The *linear address*, which is a virtual address, is mapped to a physical address.

By combining these two memory management techniques, worlds where the same address can resolve differently across worlds can be created as follows:

- Each world is backed by a fixed-size memory segment. Each segment occupies a large enough range of the virtual address space to hold the entire program state.
- Within a segment, object references are not represented with virtual memory addresses but with segment offsets. The same object lives at the same segment offset in all segments. Thereby, objects with identical state have identical underlying memory across all worlds.
- Every x86 instruction that accesses memory is prefixed such that it works within the current segment. The current segment is selected by setting the segment register to the segment's start address. Depending on the selected segment, a segment offset resolves to different virtual addresses.

We manage a fixed number of such worlds. When a thread attempts to start a transaction, it requests exclusive access to a world from a scheduler, sets the segment register accordingly, runs the transaction, and then releases the world again. Hence, the number of worlds determines the degree of parallelism, but is otherwise independent of the number of threads, which can change at any time.

With this process to set up worlds, we achieve the important property that identical program state is represented with identical memory across all worlds. The next step is to use that property such that worlds actually share memory.

4.2 Page Sharing

The final step in the address translation, as described above, is the mapping of virtual addresses to physical addresses. x86 CPUs divide the virtual address space into pages, blocks of contiguous virtual memory addresses. Each virtual page can be independently mapped to a page in physical memory, and this mapping can partially be controlled by a user space process. Particularly, user space processes can change the mapping of virtual pages such that several virtual pages map to the same physical page of memory. Additionally, processes can also unmap virtual pages such that they do not map to any physical page. With these mechanisms, memory sharing between worlds becomes possible.

Across all parallel worlds, an object is identified by the same segment offset. While pages technically exist in the virtual address space, we want to refer to pages within a segment's address space in a similar way. For example, the 5th page in a segment holds the same part of program state as the 5th page in another world since both pages start at the same segment offset. Henceforth, we refer to pages by their segment-local ordinal number and not by their absolute ordinal number within the entire virtual address space.

Memory segments each occupy a large, fixed-size range of the virtual address space. In other words, all memory segments consist of the same number of pages. However, these pages are initially not backed by physical memory pages; segments start out with all their pages being *unmapped*. Only if the program state grows and more memory is required, initially unmapped pages are mapped to actual physical memory. Also, for pages with the same ordinal number, if they are identical in all segments, they can all map to a single page of physical memory. We call these pages *shared* pages as opposed to *private* pages, which do not share physical memory across segments.

Figure 4 illustrates the differences between unmapped (U), shared (S), and private (P) pages. Page 0 is unmapped in both segments and thus needs no physical page. Page 1 shares one physical page across both segments. Page 2 is unmapped in one segment, but not the other. And page 3 is completely private in both segments, thus needing a physical page per segment. The mapped physical memory pages are allocated on-demand and can appear in an arbitrary order.

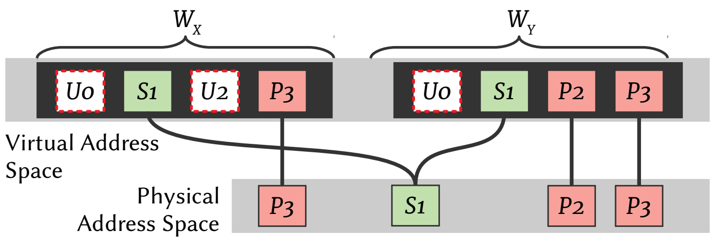

Fig. 4. Mappings of virtual pages from segments W_x and W_y to physical pages.

By solving the problem of making identical program state have identical underlying memory across segments, worlds can share state page-by-page by controlling the mapping of virtual memory pages. This page-wise sharing allows for the creation of isolated, memory-efficient worlds, without violating the expected thread and process encapsulations (**Challenge 1 & 2** of Section 2.1). Still missing, however, are the rules for when and how pages can be shared among worlds. In the following, we describe the algorithmic aspects for page-wise state management and show how the STM algorithm interacts with this implementation of parallel worlds.

5 PAGE MANAGEMENT

The memory management level discussed here should not be confused with the STM's algorithmic level: While segment memory management works with page granularity, the STM algorithm works with object granularity, i.e., write and read barriers deal with individual objects *within* these pages. The two levels have different concerns, and the approach presented here attempts to decouple them as much as possible.

In the following, we explain how the two levels interact. Specifically, the question of when a page can be shared, when it needs to be private, and when a private page can become shared again requires coordination between these levels. Hence, we look at how the transaction life-cycle influences the state of a memory segment.

5.1 Challenges

Sections 3.1ff describe the transaction life-cycle and how a commit log is used during commit and validation events for reconciling program state in parallel worlds. The commit log is a list of atomic changes to program state, each atomic change representing a so-called *revision*. Since revisions are only created through atomic commits, revisions always represent consistent program states.

Whenever a transaction validates, it moves its world forward to the most current revision. However, since validations are asynchronous with commits, worlds can stay at old revisions for an arbitrary amount of time (unless they try to commit). Hence, different parallel worlds can be at different revisions, and this disparity has consequences for the underlying memory segments and the page sharing mechanism.

The first issue can be explained with a small example: Consider an object o_A living in Page 5. Page 5 is a shared page, i.e., since o_A is identical in all segments, we map the virtual page to a single page of physical memory. Now, a transaction in world W_X modifies o_A. Hence, Page 5 needs to be made private in W_X such that the modifications do not become visible to other worlds before commit.

Later, this transaction successfully commits the changed o_A, entering a new revision in the commit log. Eventually and independently of each other, W_Y and W_Z will validate to the new revision. In W_Y and W_Z, Page 5 with o_A is still at an earlier revision. Hence, during validation, the changes that happened to o_A need to be imported to each segment's Page 5. But if we are not careful, the validation in W_Y could import changes to the shared Page 5 and thereby also affect Page 5 in W_Z. However, W_Z may not have validated yet and therefore may still be at an older revision. In that case, W_Z's Page 5 must stay at the older revision until W_Z itself validates.

> **Challenge A** Modifying a page, committing changes to a page, as well as importing changes during validation must never influence other worlds. Hence, a mechanism is needed that privatizes pages when necessary such that worlds stay fully isolated.

The second issue concerns pages that are unmapped in one segment while they may be mapped and receiving changes in other segments. For example, if o_A and its underlying Page 5 was never accessed in the segment of W_X, we may choose to keep that page unmapped in this segment. Keeping a page unmapped has the advantage that the page requires no memory and that validation does not need to import changes to that page, even if new revisions are created by other worlds. However, if suddenly a transaction in W_X accesses o_A, the right revision of that page needs to be found and mapped.

> **Challenge B** When a previously unmapped page is required, the STM must get the page's contents at the correct revision. However, there are three possible situations: 1) The requested page is mapped in another segment and is at the correct revision. 2) All other segments are at more recent revisions. 3) The requester's revision is the most recent among all segments.

In addition to the revision mismatches between worlds, the requested page may also have local modifications in all other segments. So the STM also needs a way to revert any local modifications in a page.

To address both challenges, the STM needs a way to react to page accesses before they happen, a set of rules for transitioning the mapping state of a page, and a way to reconstruct a page at any revision.

5.2 On-Demand Page Mapping

Reacting to page accesses on-demand means checking the page state before accessing an object. Since the STM already employs read and write barriers, the barriers could check the states of the object's pages and transition page states as required. However, that check would need to happen every time a transaction accesses an object, and such checks are expensive. Instead, x86 CPUs again offer an option with less overhead through *memory protection*.

Memory protection assigns each page a protection level of either *inaccessible*, *read-only*, or *read-write* (among others). If an attempt is made to read from or write to an inaccessible page, the hardware notifies the operating system, which sends a signal to the thread whose transaction attempted the access (in the case of writing to a read-only page the corresponding steps are taken). That signal can be caught by a *signal handler*, which can then react to it by adjusting the page's protection level such that the access becomes valid. The signal handler has the great advantage over using barriers that if an access conforms to the protection level, the access has no additional overhead. The cost of handling the signal is only paid once: before the protection level is adjusted.

The three protection levels *inaccessible*, *read-only*, and *read-write* match our previously used page states of *unmapped*, *shared*, and *private*. Unmapped pages have no underlying physical memory and therefore should be inaccessible; shared pages should not be modified since they may be visible in multiple segments, and they therefore should be read-only; and private pages can be freely modified and should therefore be read-write. Hence, page state transitions can be implemented

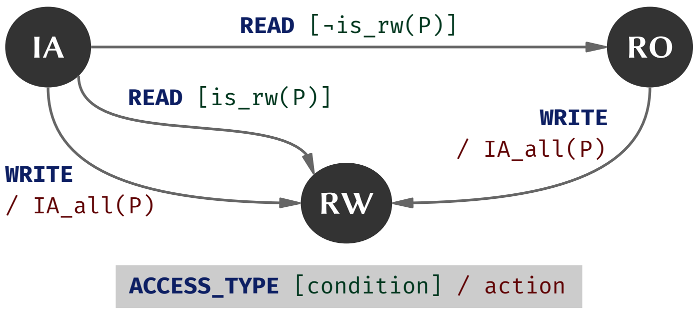

Fig. 5. Transitions between the page states inaccessible (IA), read-only (RO), and read-write (RW), as they are enforced by the signal handler.

using these protection levels and a signal handler. For example, writing to a shared read-only page should first transition the page to a private read-write page such that the write does not become visible in other segments.

Figure 5 shows the transition rules enforced by the signal handler. Generally, the signal handler's first priority is to correctly handle **Challenge A**. When necessary, pages are transitioned to the private read-write state to avoid affecting the pages of other worlds. As a second priority, the transition rules are designed to keep pages shared within the read-only state to save memory.

Once a page is read-write, it stays read-write until the STM makes an attempt to re-share pages. This re-sharing happens rarely and outside of the signal handler in a safepoint while all threads are paused. Read-only pages stay read-only until there is a write access, which transitions them to the read-write state. And inaccessible pages transition to either read-write or read-only, depending on if the access is a write or a read attempt.

However, besides the type of memory access to the page, the `is_rw(P)` condition checks if the Page `P` is currently read-write in another segment. This condition is important to uphold the following invariant: *If a page is read-only, i.e., potentially shared among several segments, the same page cannot be read-write in any other segment at the same time.* A read-only page can only co-exist with inaccessible and read-only mappings of that page (see also Figure 4, which shows all allowed combinations). The `IA_all(P)` action is responsible for making Page `P` inaccessible in all segments where the page is currently read-only.

While this invariant simplifies reasoning by reducing the number of possible states of a page across segments, the invariant seems to be a limitation. However, in reality, when a segment needs a page read-write, it means that a transaction is modifying an object in that page, and when the transaction in that segment commits, all other segments must import the changes eventually. And during import, they cannot keep their shared page as read-only anyway (see Section 5.1). So the signal handler ensures correctness by *eagerly* transitioning read-only pages of other segments to the inaccessible state.

Implementing the above page state transitions with a signal handler addresses **Challenge A** described in the previous section. For solving the second challenge, a way to reconstruct the contents of pages at specific revisions is required.

5.3 Page Reconstruction

For transitioning a page from inaccessible to read-only, the underlying memory page must be retrieved from somewhere. Since this transition is only valid if that page is not read-write in any segment, all read-only versions of that page must have the same content. Consequently, across all worlds and all revisions currently in use, the page is the same and the new read-only page can be mapped to the common shared physical page.

However, for the transition from inaccessible to read-write, there may not be an unmodified version of the page around and the page needs to be reconstructed at a specific revision. The algorithm to reconstruct a target revision rev_{target} of a Page P in segment W_{target} follows these steps:

(1) Find a suitable Page P' to copy from: Look for the segment W_y at the most recent revision that has the Page P mapped (may be at $rev < rev_{\text{target}}$).
(2) Copy Page P' from W_y to the read-write Page P in W_{target}.
(3) Revert any local modifications of segment W_y by applying W_y's recorded backup copies for Page P.
(4) If the revision of Page P' is more recent than the required revision for P, i.e., $rev > rev_{\text{target}}$, go backwards through the commit log and revert any changes that are recorded for Page P.

Step 1 may find that Page P is only mapped in a segment W_y that is at an earlier revision than segment W_{target}, i.e., $rev < rev_{\text{target}}$. This situation can arise if W_y did not validate or commit for some time, but still has the most recent page contents. Hence, the page's contents are also still valid at rev_{target} and Step 4 has nothing to do.

The algorithm sketched above addresses **Challenge B**. However, Step 4 of the algorithm has consequences that affect the mechanisms for reconciling the states of worlds, which we explain in the following section.

5.4 World Reconciliation

Step 4 of the page-reconstruction algorithm is significant because it requires that revisions can be reverted. That requirement has consequences for how the commit log and the validation really work. We previously described commit log entries as the (positive) changes that were applied to a revision. However, if the entries only recorded the new values written to memory, reverting memory to a previous revision would not be possible without keeping an infinite history. Hence, to revert changes, the commit log should record the old values, i.e., the overwritten memory.

In reality, recording old values is actually easier with the presented design, since that data is already required for aborting a transaction. The backup copies created by write barriers contain the information that needs to be recorded in the commit log entries. Hence, a commit log entry is not a list of positive changes, but a list of the backup copies recorded by the transaction that created the revision.

As a consequence, validation becomes more involved, since importing new revisions cannot be accomplished by simply applying the recorded positive changes in the log entries. Instead, for each log entry, validation needs to import the changes by copying from the actual memory pages of the segment that created the log entry. Possible local modifications of that segment again need to be reverted by applying the segment's backup copies.

With the page state transitions done using a signal handler and the ability to restore pages to certain revisions, the STM addresses **Challenge A & B** presented in Section 5.1. However, for creating opportunities to share pages, STM bookkeeping also needs careful design. The main challenge for bookkeeping is that certain metadata cannot be stored within the pages, i.e., be attached to objects. Metadata that is attached to objects could make the underlying memory look different in separate segments, even if the program state is the same.

5.5 Efficient Bookkeeping

Any attempt to write to an object is always preceded by a write barrier. That write barrier writes a flag on the object itself. Setting the flag marks the object as being recorded in the transaction's write set and as having a backup copy. The write barrier is conditional and only executes if that flag is not set on an object.

Setting the flag is a modification to the object's page and causes a page transition to the read-write state. Hence, the flag prevents sharing the page with another segment, and other segments must be able to set that flag independently. This page transition is not an issue in the case of a write barrier since the page needs to be private anyway for the upcoming modifications of that object. However, reading an object must clearly not cause such a page transition.

A transaction needs to record objects that it has read so that during validation, the set of objects recorded in a commit log entry can be compared to the transaction's read set (see Section 3.2). Indeed, it is sufficient to merely mark objects that were read. But since using a flag to mark objects would cause unwanted privatization of pages, the STM must maintain this information externally.

A straightforward solution is to maintain an actual set of read objects and make the read barrier conditionally check if an object is already in the set, and otherwise add it. But that would make the read barriers expensive. Instead, we again make use of memory segmentation: At the start of each segment, a section of memory is reserved. That section contains one byte, a *read marker*, per object in the segment. The read marker records if an object was read in the current transaction.

We define that the minimum size of an object is 16 bytes. Hence, objects can only be located at 16-byte-aligned segment offsets, and only one read marker per object is needed. The read markers section therefore is one 16th the size of the entire segment and consists only of unmapped or privately mapped read-write pages.

The read barrier's job is to mark an object as read by taking the object's segment offset, dividing it by 16, and writing a byte to that segment offset. The byte is a number that identifies the currently running transaction such that the markers only need to be reset (and unmapped) every 255 transactions in a segment. Hence, validation can check if an object was read by comparing its read marker with the current transaction's number.

This design makes the read barrier extremely cheap. The barrier consists of only unconditional instructions that can be perfectly predicted by the CPU. These low-cost read barriers are a key to reaching overall high performance with limited parallel computing resources.

6 EVALUATION

The presented STM system is a solution to efficient parallel quantum execution for a shared memory threading model. We evaluate the STM system as the runtime of a parallel Python VM. We compare the non-parallel *PyPy* VM with *PyPy-STM*. PyPy-STM is identical to PyPy, except for its use of STM for synchronization, which allows it to run several Python threads in parallel. Both VMs include the same JIT compiler [Bolz et al. 2009]. However, for PyPy-STM, the JIT additionally generates the required STM barriers and other integration code such that JIT-compiled code runs under quantized atomicity. These differences may lead to different compilation decisions by the JIT (e.g., selection of the program to be compiled) and thus may account for some of the performance difference in the comparison between the two VMs.

6.1 Host Environment

The STM is configured with $W = 8$ parallel worlds, i.e., 8 segments of memory allow for up to 8 transactions running in parallel. For PyPy-STM, this means that up to 8 Python threads can make progress at the same time, and if there are more than 8 threads, a scheduler assigns worlds dynamically. Limiting W to 8 means that the operating system never handles more than 8 active threads that compete for CPU cores. We thereby limit oversubscription if the host system has ≤ 8 cores.

The limit of $W = 8$ is a deliberate choice to adapt to the average host environment of dynamic language VMs. Since dynamic languages are not the first choice for high-performance computing, they do not find much use in these high-performance computing environments (with the exception of scripting). Rather, they are used in commodity desktop computers that usually exhibit between two and eight processor cores. Exhibiting good performance in a low-core environment is usually not a strong point of STM systems [Cascaval et al. 2008; Dragojević et al. 2011]. However, for dynamic language programs, it is essential that parallelization efforts pay off in those environments, and we therefore cannot depend on the availability of tens of processor cores.

For the evaluation, we use two systems: System A has 512 GB of memory and two Intel® Xeon® E5-2699v4 CPUs (released in 2016), each with 22 cores. But for the aforementioned reasons, all benchmarking is limited to a single CPU with 22 cores and 256 GB of memory (one NUMA node). System B has 64 GB of memory and four Intel® Xeon® E7-4830 CPUs (released in 2011), each with 8 cores. Again, we limit all benchmarking to a single CPU with 8 cores and 16 GB of memory.

6.2 Benchmarks

The main goal of the evaluation is to get an understanding of PyPy-STM's performance, overhead, and potential. To this end, we use a collection of multi-threaded Python programs [PyPy 2018] and measure their performance on PyPy and PyPy-STM. All benchmarks run unmodified on PyPy as well as on PyPy-STM, and each can be configured to distribute its workload on any number of threads. The concurrency pattern of each of these programs generally follows the fork-join pattern, and the computation in each thread is largely independent of computations on other threads — with the exception of *btree* and *skiplist*.

The two benchmarks *btree* and *skiplist* are Python programs that insert, remove, and check for elements in a shared data structure. A configurable number of threads each performs a number of these operations, which lead to contention on the data structure. PyPy and PyPy-STM fundamentally differ in the way they synchronize the shared data structure: PyPy simply performs each operation in a critical section and thereby serializes them. PyPy-STM, however, synchronizes by making each operation transactional. Hence, PyPy cannot scale with multiple threads, and PyPy-STM is exercised in a scenario of systematic contention on a shared data structure.

The evaluation of Meier et al. [2016] is based on the same set of benchmarks. The execution model and the implementation are unchanged, and the evaluation was done on the same System B. However, the earlier evaluation differs from the one presented here in the following ways:

- In the earlier evaluation, the JIT was disabled to avoid a discussion of the JIT's influence on the results. In the current evaluation, the JIT is enabled and its influence discussed.
- Since the JIT vastly improves the execution times of all benchmarks, we increased the benchmarks' workloads (input sizes) for this evaluation.
- The STM system received changes since the earlier evaluation. These changes cause small improvements to the execution times of most benchmarks.

6.3 Overall Performance

Table 1 shows the execution time for the set of benchmarks, obtained on System A. The reported numbers are averages of the measured execution times of 30 benchmark iterations. Iterations are distributed over 5 fresh VM instances, each instance performs a few iterations to warm up the JIT, and then we measure 6 warm iterations.

Table 1. Execution time (sec) of benchmark iterations measured on System A, reported as the mean and the standard deviation of 30 iterations (standard deviations below 10% are omitted). Highlighted in **bold** is the best execution time for a given benchmark and VM. The last column reports the highest speedup achieved by PyPy-STM relative to the best PyPy configuration.

Python VM	**PyPy**				**PyPy-STM**				**Speedup**
Threads	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**	*
regex-dna	**10.09**	12.34	14.02	14.70 ±1.8	11.56	5.80	3.02	**1.65**	6.10×
nqueens	**1.73**	3.04	3.13	3.12	2.90	1.68	0.99	**0.66**	2.63×
mersenne	**28.87**	29.93	31.26	32.30	41.54	22.67	11.62	**6.15**	4.69×
mandelbrot	11.12	**9.49**	19.21	20.32	9.91	4.96	2.61	**1.38**	6.87×
fannkuch-redux	**31.35**	65.89	65.67	65.79	46.68	23.46	12.01	**6.20**	5.05×
k-nucleotide	**5.66**	8.30	8.66	8.20	9.37	5.97	4.56	**3.90**	1.45×
perlin	**17.81**	19.75	32.26	36.58	17.23	9.83	6.30	**5.77**	3.09×
richards	28.23	**26.01**	49.42	60.86	34.50	18.08	9.30	**4.80**	5.42×
raytrace	**12.95**	15.66	27.06	28.70	17.99	11.68 ±2.5	5.70 ±0.7	**4.31** ±0.5	3.00×
parsible-bench	**5.21**	7.24	7.70	9.39	9.14	5.91	4.23	**3.14**	1.66×
btree	**4.72**	5.31	11.02	81.47	6.30	4.99	4.21	**2.64**	1.79×
skiplist	**5.26**	5.93	11.41	31.25	7.11	5.00	3.84	**2.98**	1.76×

As expected, the execution time of benchmarks does not improve on PyPy as more threads are available since PyPy is not a parallel VM and uses a GIL for synchronization in a multi-threaded scenario. The majority of benchmarks therefore achieves the best result on a single thread. (Two benchmarks, *mandelbrot* and *richards*, see a minor improvement with 2 threads.) However, we also observe that some benchmarks, e.g., *richards* and *raytrace*, display a severely higher execution time with multiple threads compared to the single-threaded execution. The reason for that slowdown for PyPy is increased contention on the GIL, and hence, increased synchronization overhead. For PyPy, we conclude that threads should be avoided if possible, since the use of threads is likely to slow down the overall program execution. This conclusion may also apply to other VMs developed with the RPython framework since the GIL is present in all of them.

The results for PyPy-STM show good scalability. The execution time of all benchmarks decreases with an increasing number of threads. In all cases, PyPy-STM achieves the best performance when the largest number of threads is available. The last column of Table 1 reports the highest achieved speedup of PyPy-STM relative to PyPy; this figure is the ratio of the best execution time for each VM. The speedups range between 1.5× and 6.9×, for 8 threads in PyPy-STM. These results indicate that parallelization efforts based on threads can yield significant speedups on PyPy-STM over PyPy. The same speedups may also be observed by other VMs that use the RPython framework and that switch to using the presented STM for synchronization instead of the GIL.

6.4 Background Tasks and STM Overhead

Figure 6 contrasts the execution times of PyPy-STM with PyPy per thread configuration. The single-thread configuration shows the overhead introduced by the STM instrumentation with speedup factors below 1. On multiple threads, we shed light on a scenario where a program must use threads. For example, a program may start tasks on background threads while the main thread handles the user interface. In this scenario, threads are necessary to keep the user interface responsive.

With few exceptions (*mandelbrot*, *perlin*), PyPy-STM shows a significant slowdown on a single thread compared to PyPy. The reason for this overhead is the additional STM instrumentation and support code, which are enabled even if only a single thread runs in the VM. Avoiding the STM overhead in a single-threaded execution is an area of active research. Possible solutions

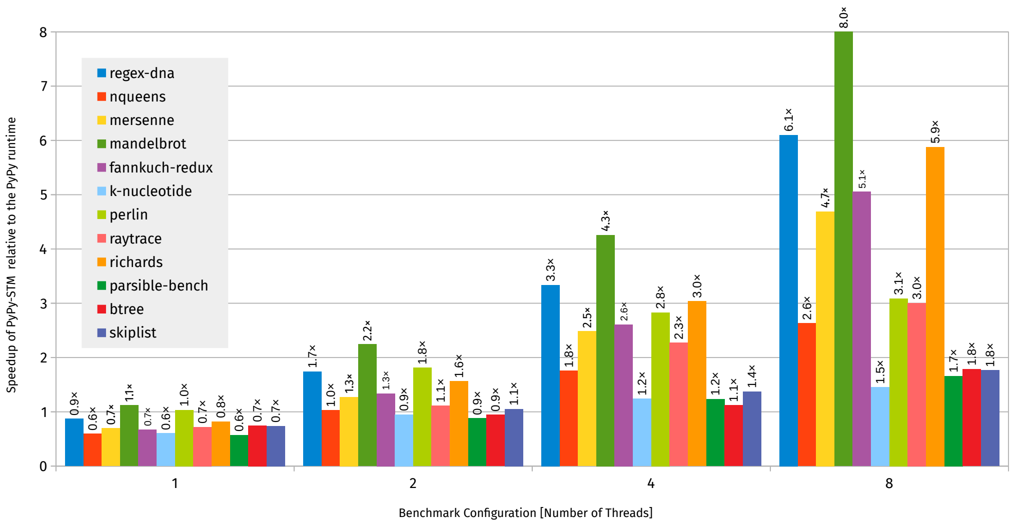

Fig. 6. Speedups of running the benchmarks on PyPy-STM relative to PyPy per thread configuration. Execution times were measured on System A.

Table 2. CPU time overhead caused by aborted transactions, in percent of CPU time of committed transactions. The numbers are averaged over 5 VM instances; standard deviations below 10% are omitted. Cases with ≥ 10% overhead are highlighted in **bold**.

Threads	**1**		**2**		**4**		**8**	
regex-dna	0.1%		0.5%		1.2%		3.5%	±1.6
nqueens	0.4%	±0.1	0.4%	±0.1	1.0%	±0.4	0.8%	±0.8
mersenne	0.0%		9.4%	±1.0	**11.2%**		**12.0%**	±1.6
mandelbrot	0.0%	±0.0	0.1%	±0.0	0.1%	±0.1	0.5%	±0.4
fannkuch-redux	0.0%		0.0%	±0.0	0.9%	±0.3	1.2%	±0.6
k-nucleotide	0.0%		3.7%	±0.9	1.4%	±0.2	0.9%	±0.2
perlin	0.0%		0.1%	±0.0	0.5%	±0.1	2.1%	±2.1
richards	0.0%		4.7%	±5.8	4.3%	±3.9	4.1%	±3.0
raytrace	0.1%	±0.1	0.1%	±0.1	0.2%	±0.1	0.0%	±0.0
parsible-bench	0.3%	±0.0	0.2%		0.3%		0.6%	±0.2
btree	0.0%		**11.7%**		**23.6%**		**50.2%**	
skiplist	0.0%		**20.0%**		**52.8%**	±6.0	**116.4%**	±12.2

include improving the STM's single-thread mode (similar to the work of Wamhoff et al. [2013]) and employing the JIT to avoid instrumentation when no concurrent thread is active.

With two or more threads available, the speedups achieved by PyPy-STM over PyPy become significant. In the case of *mandelbrot* (a moderately meaningful but nonetheless widely-used benchmark) running with two threads on PyPy-STM is 2.2× faster than running it with two threads on PyPy. Again, these results support the conclusion that using threads on PyPy will often degrade the overall performance. The performance advantage of PyPy-STM over PyPy can indeed be significant in the background task scenario.

6.5 Speculation Overhead

Transactional memory is an optimistic concurrency control mechanism. Speculatively running transactions in parallel means that speculation may sometimes be too optimistic. If speculation fails, all the work that was speculatively executed is lost. Hence, it is informative to look at the percentage of CPU time that is lost in aborted transactions. Table 2 presents the ratio of CPU time spent by aborted transactions to CPU time spent by committed transactions. Hence, the ratio is a measure of overhead. For example, 100% means that half of all CPU time spent over all threads was lost by aborting transactions.

Aborted transactions have two main causes: First, benchmarks with systematic contention (*btree*, *skiplist*) inherently cause conflicts, and therefore transaction aborts. Second, the decision of when to commit (or abort) a transaction is made by the STM system. Deciding to commit a transaction too late can lead to more transaction aborts, and committing a transaction early can *avoid* a transaction abort. The STM system has to decide on a trade-off: Short transactions lead to fewer conflicts, but they cause more frequent commits, which lead to a higher STM overhead. Long transactions lead to fewer commits and thus a lower STM overhead, but they increase the probability of conflicts.

Table 2 reflects the systematic contention problem: *btree* and *skiplist* show large percentages of lost CPU time. Indeed, with 8 threads, *skiplist* spends more time in aborted transactions than doing useful work in committed transactions. Both benchmarks suffer from a high degree of systematic contention on the shared data structure. All other benchmarks usually show percentages below 10% (with the exception of *mersenne*). Indeed, the low percentages reported are still largely caused by systematic conflicts within the framework for distributing the work onto multiple threads.

We conclude that a percentage of less than 10% in benchmarks with independent computations is acceptable. Hence, the decisions of the STM system appear to handle this case well enough. For the two benchmarks with systematic contention, we may conclude that a higher percentage is acceptable. However, once we reach 50%, meaning a third of the whole CPU time is spent in aborted transactions, it is likely that the STM system is too optimistic in its decisions and should shorten the transactions to avoid aborts. Future research should aim at better understanding this trade-off to help the STM system make better decisions in the case of high contention.

6.6 Scalability and the World Limit

Figure 7 shows PyPy-STM's scalability (in the number of threads (T) available) and illustrates what happens if the number of threads exceeds the number of parallel worlds (W) in the STM. W limits the number of transactions that can run in parallel. To run a transaction, a thread requests access to a world from a scheduler, and hence, W also limits the number of parallel active threads. We therefore expect the maximum speedup to be achieved if $T = W$.

Indeed, we observe that the majority of benchmarks speeds up with an increasing T until $T = W$, with the exception of *perlin*. *perlin* reaches a peak in performance already on $T = 7$ threads and its performance declines afterwards. Interestingly, the highest peak in performance for this benchmark is at $T = 14$. On $T = 7$ threads and on $T = 14$, the measured execution times are 4.63 ± 2.10 and 3.80 ± 2.64 respectively. These exceptionally high standard deviations are caused by the decisions of the JIT compiler.

perlin contains a loop with numerous branches, and such code is generally problematic for a JIT like PyPy's [Bolz et al. 2009], which is a *tracing* JIT. Tracing JITs, as opposed to method JITs, record and compile execution traces through a program instead of compiling the program method-by-method. A loop with a lot of paths requires substantial tracing, and the first recorded trace can greatly influence the performance of the compiled code. By introducing parallelism, the JIT's decision of what to trace becomes dependent on the schedule of tasks at runtime. And with

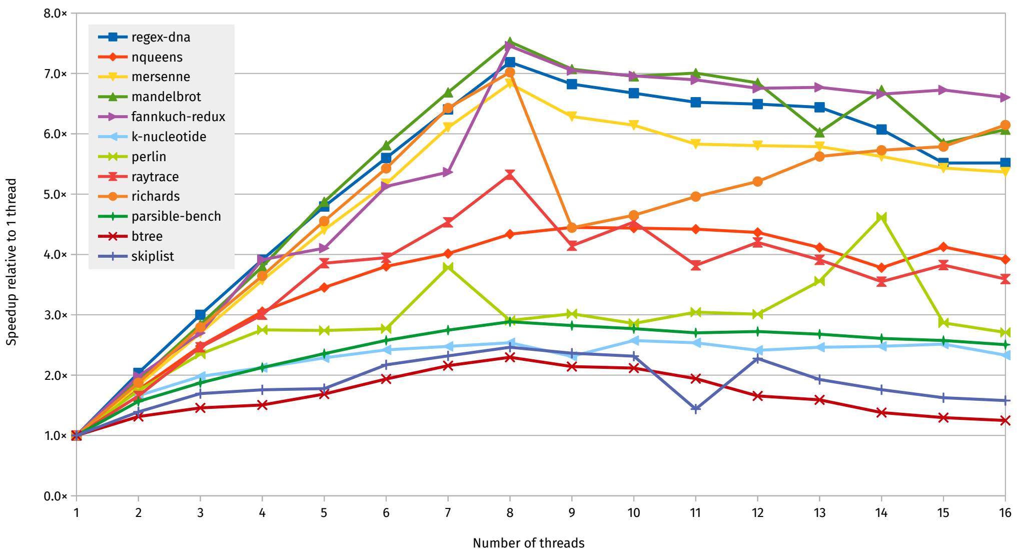

Fig. 7. Scalability per benchmark on PyPy-STM with an increasing number of threads. Measurements were made on System A (22 cores available) with the number of parallel worlds fixed to $W = 8$.

Table 3. Execution time on System B. Compare with Table 1

Python VM	**PyPy**				**PyPy-STM**				**Speedup**
Threads	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**	*
regex-dna	**30.45**	30.67	30.89	30.94	34.73	17.81	8.93	**4.50**	6.76×
nqueens	**2.73**	2.77	2.77	2.84	5.44	2.81	1.56	**1.18**	2.32×
mersenne	**55.02**	55.04	55.07	55.13	67.90	37.69	19.29	**10.37**	5.31×
mandelbrot	**18.81**	19.43	19.23	19.15	19.02	9.68	4.89	**2.51**	7.50×
fannkuch-redux	61.44	**61.37**	61.45	62.65	85.57	45.89	22.70	**17.89**	3.43×
k-nucleotide	10.16	10.18	**9.97**	9.99	16.10	10.06	7.04	**6.10**	1.63×
perlin	**36.14**	38.03	36.87	37.98	37.38	20.36	12.07	**7.73**	4.68×
richards	**50.18**	51.19	51.25	50.39	62.84	33.83	18.38 ±2.6	**9.05**	5.54×
raytrace	**23.91**	26.44	26.66	28.38	33.70 ±7.8	22.42 ±4.8	**13.32** ±2.6	22.89 ±7.7	1.79×
parsible-bench	9.60	**9.55**	9.91	9.74	16.44	9.78	6.19	**4.36**	2.19×
btree	**7.66**	8.17	9.27	14.80	10.06	8.46	5.74	**4.21**	1.82×
skiplist	**8.30**	8.50	8.82	11.94	10.48	7.83	5.56	**4.25**	1.95×

$T = 7$ and 14, the schedule often hits a sweet spot that makes the JIT trace the best paths first more often than not.

When T exceeds W, the performance of most benchmarks degrades continuously. The performance degradation is expected since distributing the same amount of work onto more and more threads increases the overhead of starting, managing, and synchronizing these threads. But there is again one exception: *richards* sees a significant drop in performance going from $T = W$ to $T = W + 1$ and later recovers again. From these results, we conclude that the scheduler, which is responsible for assigning worlds to threads, handles the situation of $T > W$ reasonably well in many cases, but in the worst case the scheduler is responsible for a significant performance penalty.

6.7 Comparing Host Systems

For comparison, we also evaluated the two VMs on System B. Although an older hardware platform, System B allows an interesting observation regarding runtime system configuration. On System A and System B, the best achieved speedups are comparable and range between 1.6× to 7.5×. However, putting Table 1 side-by-side with Table 3 makes a difference in behavior for PyPy stand out. On System B, the PyPy execution times of the benchmarks remain mostly unaffected by an increase in the number of threads; on System A, however, we see severe slowdowns in some of the benchmarks when comparing the 8 thread execution time to the single-threaded execution time.

An explanation for this difference is given by the rate at which the GIL is acquired and released across the threads. In PyPy, the time for which the GIL is held before being yielded to another thread depends roughly on the number of instructions executed. The higher the rate of instructions executed, the higher the rate of GIL acquire and release operations. Since System A is faster than System B (geometric mean of 1.9× in single-threaded execution), the rate of GIL yields is higher, and a higher rate means higher relative synchronization overhead. Indeed, if we forcibly *reduce* the rate by increasing the number of instructions to be executed before the GIL is yielded, the execution time on 8 threads approaches the level of the single-threaded execution time.

Naturally, the GIL yield rate is a trade-off between the level of concurrency and synchronization overhead. With a higher yield rate, all threads can make progress more often, which can also be important for the responsiveness of a system. But with a lower yield rate, the synchronization overhead of acquiring and releasing the GIL is lower. However, a fixed yield rate cannot be optimal at the same time on slow and on fast systems. Hence, an approach that adapts the GIL yield rate to the host system's performance is necessary for PyPy.

Another interesting data point on System B is the performance of *raytrace* on PyPy-STM: On System B, the 8-threaded execution time is higher than the one on 4 threads. However, we also observe that *raytrace*'s execution time shows unusually high standard deviations, a fact we also observe on System A. Again, the cause of these high standard deviations is the JIT. This benchmark is very sensitive to the decisions of the JIT compiler. Depending on the decisions made, the highest execution time can be twice as high as the lowest execution time. Generally, JIT decisions are made non-deterministically, which is also true for PyPy. However on PyPy-STM, we introduce additional non-determinism through parallelism and thereby accentuate the problem. Hence, measuring *raytrace* again can either show good scaling similar to System A, or produce even worse results than those currently shown for System B.

7 RELATED WORK

To remove the GIL in Python and Ruby implementations, previous work [Odaira et al. 2014; Riley and Zilles 2006; Tabba 2010] focused on directly using hardware transactional memory (HTM), which is built into a CPU. Fully hardware accelerated transactional memory promises low overhead. However, current HTM implementations strictly limit the amount of memory a transaction can access before overflowing a hardware buffer and requiring the transaction to abort. This limit proves to be a problem since our approach requires fitting at least one quantum in one transaction. PyPy clearly favors performance over memory consumption and hence produces quanta that access multiple megabytes of memory, which more often than not exceeds the limits of current HTM implementations. The HTM approach requires better HTM implementations that offer a software fallback mechanism when buffers overflow.

Software transactional memory (STM) systems are steadily improving. In our context, work that focuses on lowering the overhead of STM in few-core environments is the most relevant [Dalessandro et al. 2010; Wamhoff et al. 2013]. In particular, integrating the master and helper design of Wamhoff et al. [2013] may be a solution to the single-thread overhead that we measured on PyPy-STM.

Another development for STM has been the use of memory protection for providing strong atomicity in the interaction of transactional and non-transactional code [Abadi et al. 2009]. However, with the quantized atomicity of Parallel RPython's execution model, all code is part of a quantum and therefore part of a transaction. We do not have interactions between transactional and non-transactional code. Instead, we use memory protection for on-demand memory page management and thereby ensure safe memory sharing between worlds.

The approach of slow-path barricading [Swaine et al. 2010] allows *safe* code of a VM to execute in parallel, and reverts to sequential execution when *unsafe* code is encountered. With this design, a sequential VM can be parallelized incrementally. In the realization of Parallel RPython's execution model, the STM fulfills a similar role: It detects and resolves conflicts for *unsafe* code automatically. Hence, slow-path barricading is a manual, precise, and incremental approach to parallelise an existing VM, and the use of STM is an automatic, optimistic, and complete method to parallelise RPython-made VMs.

Recently, an approach to a thread-safe object model for dynamic languages was proposed [Daloze et al. 2016] in the context of Truffle [Wimmer and Würthinger 2012; Würthinger et al. 2012]. The object model avoids certain thread-safety issues, such as lost-field updates and out-of-thin-air values, and does that with a low cost to performance. An extension [Daloze et al. 2018] to this approach enables gradually increasing levels of synchronization for built-in collection types. With the extension, VM developers can implement synchronization for single-collection operations, again with low impact to performance.

We approach the problem of thread-safety in parallel VMs from a different direction. Instead of solving the problems one after another, our approach starts with safety by default and the complete GIL semantics, and with the possibility to weaken the safety guarantees in individual cases. As such, our approach provides thread-safety and arbitrary atomic operations without the need for manual synchronization. But our approach has noticeable overhead where Daloze et al. [2016, 2018] show a low performance impact.

The concept of transactional worlds bears some similarity to transactional *orthogonal persistence*. Earlier, Atkinson et al. [1982], Albano et al. [1985], and Dearle [1988], to name but a few, designed programming languages that make persistence transparent to the programmer, the same way as our parallel worlds are transparent to the dynamic language. Orthogonal persistence was even integrated into operating systems [Liedtke 1993]. And more recently, the availability of non-volatile RAM spurs research into whole-system persistence [Narayanan and Hodson 2012].

Persistence deals with failure models, and depending on transactional semantics is one way to recover from failures. Orthogonal persistence in a programming language can be thought of as a single world, with transaction aborts and retries, but without the need to reconcile multiple worlds. With parallel worlds, we use transactions for speculative parallel execution, not for failure recovery. And we maintain consistency across several such worlds that each hold a revision of the entire program state.

8 CONCLUSIONS

Dynamic languages pose great challenges to the implementation of VMs that supports the execution of parallel programs. Important languages like Python and Ruby employ the GIL memory model (GMM), which guarantees sequential consistency and instruction atomicity. These guarantees of the GMM are difficult to uphold efficiently by a parallel VM.

We present a first implementation of the Parallel RPython execution model, which facilitates the implementation of memory models as strong as the GMM. This execution model serves as the base of the VM, which may host a program written in one of several dynamic languages. The model's implementation must therefore be language agnostic, cannot break established expectations of the languages, and must address the needs of all VM components. In particular, we highlight the importance of providing the abstraction of *parallel worlds* for the JIT compiler. Worlds decouple parallel activities and allow concurrent threads to proceed without interference until a suitable point is reached. Without this world isolation, the JIT would need to handle concurrent changes and protect against inconsistent state.

Parallel worlds are directly supported by a custom STM system. Whereas many STM systems aim to support individual objects or data structures, here the STM provides separation and isolation to worlds that each may be as large as the complete program state of a process. The STM system leverages the architecture capabilities of the x86-CPU (a common platform in many scenarios) to provide an efficient implementation. It uses the built-in memory segmentation feature of x86 CPUs for separating worlds and employs virtual memory to manage page-wise memory sharing across worlds. Memory sharing is controlled by reacting to memory accesses on-demand with the help of the CPU's memory protection and a signal handler. And page state transition rules allow the signal handler to map and share pages correctly. Through this design, memory page management is largely decoupled from the STM algorithm. The resulting STM system builds on this synergetic combination of hardware features and software techniques that implement parallel worlds efficiently.

The evaluation of PyPy-STM (our parallel VM) compares to PyPy (a GIL-based Python VM) and shows that PyPy-STM achieves significant speedups over PyPy-GIL. On a platform with 8 cores, the achieved speedups are in the range of 1.5× to 7.5×. The evaluation also shows that PyPy cannot benefit from additional parallel computing resources in the system, instead it often slows down programs due to high synchronization overhead.

Several benchmarks display a sensitivity to the decisions of the JIT compiler. The performance of these benchmarks is more unstable on PyPy-STM than on PyPy. The reason for this instability is the additional source of non-determinism introduced by parallelism. While the JIT on PyPy is also not fully deterministic, introducing parallelism makes these decisions dependent on the schedule of threads at runtime. In other words, the added non-determinism accentuates a problem of PyPy's implementation of a tracing JIT. So while the problem is more visible on PyPy-STM, it not only shows up in the context of parallel computation and should therefore be considered a separate topic of future research.

There are ample opportunities for future research to improve on what we presented: The Parallel RPython model provides strong guarantees by itself and thereby requires a powerful synchronization mechanism such as transactional memory. But many dynamic languages do not depend on a GMM, and they might benefit from a selective weakening of the model such that synchronization becomes cheaper. Additionally, there are various directions in which the STM itself could be improved, such as avoiding the STM overhead for single-threaded execution. But overall, the presented approach lets the Parallel RPython execution model meet the challenges of dynamic languages. As developers of parallel VMs for dynamic languages need to exploit the benefits of parallel execution, Parallel RPython now provides a solid foundation.

ACKNOWLEDGMENTS

We thank our reviewers for their comments, which pointed out several weaknesses of the early paper. The comments helped greatly and strengthened the paper in many ways. We also thank Michael Faes and Aristeidis Mastoras for the numerous enlightening discussions.

REFERENCES

Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 185–196. https://doi.org/10.1145/1504176.1504203

Antonio Albano, Luca Cardelli, and Renzo Orsini. 1985. GALILEO: A Strongly-typed, Interactive Conceptual Language. *ACM Trans. Database Syst.* 10, 2 (June 1985), 230–260. https://doi.org/10.1145/3857.3859

Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910

Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. 1982. PS-algol: An Algol with a Persistent Heap. *SIGPLAN Not.* 17, 7 (July 1982), 24–31. https://doi.org/10.1145/988376.988378

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827 00124.

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? *Queue* 6, 5 (Sept. 2008), 40:46–40:58. https://doi.org/10.1145/1454456.1454466 00232.

Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. 2010. Transactional Mutex Locks. In *Euro-Par 2010 - Parallel Processing (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 2–13. https://doi.org/10.1007/978-3-642-15291-7_2

Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 642–659. https://doi.org/10.1145/2983990.2984001

Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2018)*. ACM.

Alan Dearle. 1988. *On the Construction of Persistent Programming Environments*. Ph.D. Dissertation. AAIDX86382.

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. https://doi.org/10.1145/1924421.1924440

IronPython. 2018. The IronPython Project. http://ironpython.net/.

JRuby. 2018. Concurrency in JRuby. https://github.com/jruby/jruby/wiki/Concurrency-in-jruby.

Jython. 2018. Jython Concurrency. http://www.jython.org/jythonbook/en/1.0/Concurrency.html.

Robert R. Kessler and Mark R. Swanson. 1990. Concurrent Scheme. In *Proceedings of the US/Japan Workshop on Parallel Lisp: Languages and Systems*. Springer-Verlag, Berlin, Heidelberg, 200–234. http://dl.acm.org/citation.cfm?id=646454.693125

Johann M. Kraus and Hans A. Kestler. 2009. Multi-core Parallelization in Clojure: A Case Study. In *Proceedings of the 6th European Lisp Workshop (ELW '09)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/1562868.1562870

Jochen Liedtke. 1993. A persistent system in real use — experiences of the first 13 years. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*. IEEE, 2–11.

Yu David Liu, Xiaoqi Lu, and Scott F. Smith. 2008. Coqa: Concurrent Objects with Quantized Atomicity. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 260–275.

Cristian Mattarei, Clark Barrett, Shu-yu Guo, Bradley Nelson, and Ben Smith. 2018. EMME: A Formal Tool for ECMAScript Memory Model Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 55–71.

Remigius Meier, Armin Rigo, and Thomas R. Gross. 2016. Parallel Virtual Machines with RPython. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/2989225.2989233

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2150976.2151018

Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 2014. Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/2555243.2555247 00005.

PyPy. 2018. PyPy Benchmarks on Bitbucket. https://bitbucket.org/pypy/benchmarks.

Python Software Foundation. 2018. Python FAQ. https://docs.python.org/2/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe.

Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

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)*. ACM, New York, NY, USA, 998–1008. https://doi.org/10.1145/1176617.1176758 00013.

James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. 2010. Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 583–597. https://doi.org/10.1145/1869459.1869507

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. https://doi.org/10.1145/1978907.1978911 00007.

Laurence Tratt. 2009. Dynamically Typed Languages. *Advances in Computers* 77, Jul (2009), 149–184.

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 *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 113–122. https://doi.org/10.1145/2442516.2442528 00003.

Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723 00012.

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587