Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution

Nicholas Riley   Craig Zilles

Department of Computer Science
University of Illinois at Urbana-Champaign
{njriley,zilles}@uiuc.edu

Abstract

Lightweight dynamic language runtimes have become popular in part because they simply integrate with a wide range of native code libraries and embedding applications. However, further development of these runtimes in the areas of concurrency, efficiency and safety is impeded by the desire to maintain their native code interfaces, even at a source level. Native extension modules' lack of thread safety is a significant barrier to dynamic languages' effective deployment on current and future multicore and multiprocessor systems. We propose the use of hardware transactional memory (HTM) to aid runtimes in evolving more capable and robust execution models while maintaining native code compatibility. To explore these ideas, we constructed a full-system simulation infrastructure consisting of an HTM implementation, modified Linux kernel and Python interpreter.

Python includes thread constructs, but its primary implementation is not architected to support their parallel execution. With small changes, a runtime can be made HTM-aware to enable parallel execution of Python code and extension modules. We exploit the semantics of Python execution to evaluate individual bytecodes atomically by default, using nested transactions to emulate programmer-specified locking constructs where possible in existing threaded code. We eliminate common transactional conflicts and defer I/O within transactions to make parallel Python execution both possible and efficient. Transactions also provide safety for foreign function invocations. We characterize several small Python applications executing on our infrastructure.

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

General Terms Design, Languages

Keywords Transactional Memory, Dynamic Languages, Python, Concurrency, Locking, Safety

1. Introduction

Mainstream runtimes for lightweight dynamic languages including Perl, Python, Ruby and Tcl have been successful in part because they easily interface with native code, through extension modules and by embedding themselves into host applications. These runtimes' native code interfaces, like the runtimes themselves, are simple, easy to understand, transparent and portable. By placing few restrictions on what an extension module or embedding application can do, they function as "glue" in integrating disparate code bases.

Unfortunately, widespread use of open-ended native code interfaces restricts a runtime's ability to evolve concurrency, efficiency and safety, which in turn can impair the language's applicability. A multithreaded host application embedding a thread-unsafe dynamic language runtime may not scale well on current and future multicore and multiprocessor systems. Alternative implementations of these languages, built upon heavyweight runtimes such as Java or .NET, already scale to multiple processors, can outperform the mainstream implementations, and support safer execution, but are infrequently used. Disadvantages of these implementations include increased memory overhead and startup time, restricted portability, embeddability and extensibility.

We propose addressing several issues impeding development of lightweight dynamic language runtimes with hardware transactional memory (HTM) mechanisms. A HTM extends a machine's processor and memory architecture to support user-controlled speculative execution, conflict detection, and related facilities. We use features of a proposed HTM to incrementally incorporate concurrent execution and improved safety in a Python runtime, without significantly complicating the runtime's implementation or requiring extension modules and embedding applications be rewritten.

The "official" and most popular Python runtime is CPython, a bytecode interpreter written in C; runtimes for Java and .NET also exist. The PyPy project [20] aims to automatically generate a range of next-generation Python runtimes, including interpreters and just-in-time compilers, from descriptions specified in a subset of the Python language [24]. We selected the most mature PyPy target: `pypy-c`, an interpreter compiled from generated C code. Its design is similar to CPython's, and the techniques we present would apply with a little more work to CPython.

The PyPy and CPython runtimes implement primarily non-concurrent threading using OS-level threads. Both use a Global Interpreter Lock (GIL) to prevent two threads from concurrently interpreting Python bytecode. A thread yields control by releasing the GIL between bytecodes, or before a blocking I/O operation [7].

We take a first step to transactional concurrency for Python by constructing a full-system prototype for hardware transactional execution, enabling PyPy to run existing lock-synchronized, GIL-threaded Python code in parallel, and falling back to sequential execution where required. Specifically, this paper makes the following contributions:

First, we propose a method for *safe lock-transaction coexistence*, in which threads using locks and transactions for concurrency control can enforce the same set of atomicity constraints.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

Common embedding environments, such as the Apache HTTP Server's mod_python and graphical or other event-based applications, allow Python execution in event handlers. Embedding applications' threading models usually differ from the Python runtime's; as a result, applications must carefully manage the context in which Python code is executed and avoid deadlock when Python code accesses data structures in the embedding application. Our model permits nontransactional code, such as that in an unmodified embedding application, to execute in the same address space, and in parallel with transactional code.

Second, *automatic transactions* support transactional execution of extension module functions that perform I/O, by stopping and starting transactions around I/O operations within a single bytecode execution, matching the most common GIL usage pattern in Python extension modules. Extension modules vary widely in their thread safety, potentially introducing bugs and incompatibilities in languages such as Perl whose runtimes now permit concurrent execution. By using hardware transactions, extension modules written without thread safety in mind can continue to work.

Python, like other dynamic languages, has evolved generic foreign function interfaces (FFIs) which offer the dynamic language programmer support for invoking arbitrary native functions and managing native data structures, without manually wrapping them in a native language—a tedious and repetitive process. While generic FFIs can lower the barrier to native code integration and give dynamic language programmers opportunities to compromise the runtime's stability, they also increase the ability of the runtime to introspect native code execution. Extension modules written using generic FFIs benefit from increased transparency, exposing to the runtime marshalling and exception handling that would otherwise be hidden in native code.

By enclosing individual native function calls in transactions, we can, in many cases, increase concurrency and safety. We propose a mechanism for *optimistic deferral* of side effects to the end of an enclosing transaction. A key property of transactional memory systems is composability: transactions whose behavior is not externally visible can be arbitrarily nested. With optimistic deferral, we can extend this composability to extension modules that include I/O. We also describe the use of transactions to insulate the runtime from programming errors and crashes caused by native function calls.

This paper is organized as follows. In Section 2, we give background on both the challenges with introducing concurrency into dynamic languages and research on transactional memory. In Section 3, we discuss both the benefits of using transactions for Python execution and the runtime changes required. Specifically, Section 3.1 introduces the manner in which transactions enable concurrency while maintaining the semantics of Python's global interpreter lock-based threading model. Section 3.2 discusses changes to the PyPy runtime which avoid false conflicts. Section 3.3 describes an execution model for running existing lock-based parallel Python applications in any combination of transactional and non-transactional threads, and Section 3.4 discusses the interactions of memory allocation and garbage collection with transactions. Sections 3.5 and 3.6 describe methods for processing and deferring non-undoable actions such as I/O within transactions. Finally, Section 3.7 discusses a simple form of protection which can guard against erroneous native code execution, and Section 4 presents a characterization of Python executing transactionally.

2. Background

In this section, we introduce the range of lightweight dynamic language concurrency models, provide a brief introduction to the capabilities of hardware transactional memory, and describe the layers of our transactional memory infrastructure below the PyPy runtime.

2.1 Dynamic Languages and Concurrency

Lightweight dynamic languages such as Perl, Python and Ruby are defined by their original, and still most commonly used, implementations. None were initially designed to support concurrent execution. While the languages' users have come to accept relatively low performance, they do expect the runtime's speed to scale with the rest of their applications. As these languages' rising popularity accompanies the emergence of mainstream systems whose primary speed gains derive from increasing the number of processor cores, their continued viability depends on implementing practical models of concurrency.

Concurrent multithreaded execution of arbitrary dynamic language code requires explicit support from extensions and embedding applications to avoid deadlocks, data corruption and other forms of incorrect execution. In addition, language runtimes' concurrency models can interact with the concurrency models of embedding applications or frameworks exposed through extension modules in unexpected ways.

For example, Ruby, Perl and Python have by now adopted one or more threading models, though no two languages share a common model. Ruby threads execute cooperatively in a single OS-level thread, with nonblocking I/O managed transparently to the user [29]. Perl's *ithreads* map to OS-level threads, but depart from the traditional threaded memory model: each thread receives a copy of global data, and shared global data is treated as the exception rather than the rule [5, 27]. This mechanism permits generalized parallel execution and improves compatibility with existing code, at the cost of copying overhead on thread creation, additional overhead accessing data shared between threads and some extension module incompatibility.

Python uses yet another model: its high-level threading API mimics Java's, and Python threads map to OS-level threads, but the mainstream CPython runtime doesn't support parallel execution of these threads except in special cases (such as blocking I/O). Prevailing Python concurrency models utilize preemptive and cooperatively scheduled threads, events and continuations, all implemented given the limitations of a single active thread executing Python code. Currently, true parallel Python execution requires restructuring an application to execute in multiple processes, or migrating to a Python implementation built on a heavyweight runtime—with the associated compromises in portability and native code integration.

In part because of the limitations of Python threading, frameworks which multiplex a single OS-level thread are commonly utilized. Networked and graphical Python applications employ asynchronous, event-driven frameworks such as Twisted [16], or user-level "micro-threading" via *tasklets* [30] or *greenlets* [23].

2.2 Transactional Memory

Motivated by the shift toward the commoditization of multiprocessors and the problems associated with managing concurrency with locks [28], research in alternative concurrency control techniques has had a renaissance. Transactional memory (TM) [11, 14] is one such approach, where programmers delimit the boundaries of critical sections and the TM system guarantees atomic execution of these sections. In recent years, there has active research in both software (*e.g.*, [3, 8, 9, 12, 13]) and hardware (*e.g.*, [4, 18, 22]) implementations of transactional memory.

All TM systems are developed around the idea of optimistic concurrency; each transaction is executed speculatively with the expectation that it can be executed atomically. If a conflict (a read value has been written by another thread or a written value has been read or written by another thread) is detected before the transaction commits, the transaction needs to be aborted (undoing any writes by the transaction) and it must be attempted again at some later time. As a result, the three main requirements on a TM system are: 1) the ability to roll back transactions in case of an abort, 2) the ability to detect conflicts, and 3) the transaction commit is atomic. The first requirement may be implemented by *logging* old values (and their addresses) before they are overwritten in a transaction; if a transaction is aborted, these old values can be restored. The second requirement is implemented by recording the read and write sets of transactions and either eagerly (while a transaction is executing) or lazily (just before a transaction is committed) verifying that a transaction's read and write sets have not been violated. Finally, once a transaction has started committing its state, it must ensure that the commit completes without its read and write sets being violated.

In this paper, we employ a hardware transactional memory derived from the Virtual Transactional Memory (VTM) [22] proposal. Hardware-based transactional memories provide two main advantages relative to their software-based counterparts: 1) they can be implemented with negligible overhead for small transactions (those that fit in the first-level cache) and 2) they can efficiently provide strong atomicity [6], where transactions are guaranteed to be executed atomically with not only other transactions, but with non-transactional execution as well. Both of these properties are exploited in this work.

While a complete description of VTM is beyond the scope of this paper, in a nutshell, VTM is the integration of two distinct TM systems, both of which operate at a cache line granularity. Small-footprint, short-lived transactions are supported completely in hardware; the cache is used to hold the new (speculative) version of the data and the main memory serves as the log. If a transaction commits, the cached copy can be written back to memory; if aborted, the cached copy can be invalidated. Conflicts are detected using a shared-memory machine's existing invalidation-based cache coherence protocol, which guarantees that any data that is being written cannot be present in any other processor's cache. By maintaining (throughout the transaction) read permission to any read data and write permission to any written data, we can guarantee that no other processor is writing our read set and reading our write set, respectively.

For transactions that don't fit in the cache or that need to live past a context switch, VTM supports a second mode where logging and conflict detection are performed using a data structure stored in normal (non-cache) memory. This mode provides completeness, supporting arbitrary transactions albeit with lower performance. In many respects, this mode resembles a software transactional memory (STM)—except, provided the data structures are architected, many of the operations on them can be performed by hardware, much in the way that the x86's page table is walked by a hardware state machine to perform translation lookaside buffer (TLB) fills. When using this STM-like mode, we refer to a VTM transaction as having *overflowed*. When one transaction has overflowed, other potentially conflicting threads must search the in-memory data structure for conflicts. To prevent performance isolation problems (where an overflowed transaction in one process would impact the performance of another process [32]), VTM restricts transactions from spanning virtual address spaces, allowing conflict detection to be performed within a single address space.

For the experiments that we describe in this paper, we have implemented a variant of VTM through extensions to the x86 version of the Virtutech Simics full-system simulator [17] and the Linux 2.6.15.4 kernel. Our simulated system includes four physical processors (single-core Intel Pentium 4 processors without hyperthreading). The implementation extends VTM's original description [22] in several ways. Transactions may be *paused* to allow non-transactional operations (*e.g.*, I/O and system calls) within a transaction; because actions performed in paused regions are not be rolled back when a transaction aborts, a software framework for *compensation code* is provided. A conflicting transaction may be aborted and retried, or stalled until the other transaction completes [31]. A *retry* primitive supports intentional waiting [8], and *watcher* or *pseudo-transaction* support can be used in concert with transactions for memory protection. An access to a memory location watched by a pseudo-transaction invokes a signal handler, which receives information about the access and its context. The set of HTM primitives implemented by our simulator, and used by PyPy, are presented in Table 1.

3. Transactional Execution

The addition of purely software-based parallelism to CPython with fine-grained locking or software transactional memory is frequently discussed, but perenially rejected. Such techniques would complicate the interpreter implementation and require modifying Python extension modules and embedding applications for thread safety [1]. Hardware transactional memory is a resolution to this impasse: it can provide acceptable parallel multithreaded performance with small changes to the Python interpreter and no changes to extension modules or application code. In the future, transactions provide a more accessible concurrent programming model, and can provide better integration and performance when used with future transactionalized libraries and embedding applications.

We now present our modifications for transactional execution of PyPy. First, we briefly introduce PyPy's design as it affects our work, and the baseline execution model of one transaction per bytecode. To this we add support for garbage collection and operating system calls within transactions; transactionalization of, and coexistence with, lock-based code; use of transactions for safety, and optimistic deferral of operating system calls to increase concurrency.

3.1 Multithreaded Execution in PyPy

PyPy is a "Python runtime construction kit." It generates a family of runtimes by applying a series of transformations to descriptions specified in a subset of Python permitting static type inference, called RPython or "restricted Python." The primary target of PyPy is the *Standard Interpreter*, a Python interpreter mostly compatible with CPython 2.4. The interpreter is written primarily in RPython, the remainder consisting of native code that must be reimplemented for each platform. Runtimes can be generated with different target languages (including C, LLVM [15] and .NET), garbage collection strategies, and included language features such as threads and coroutines. The interpreter description can currently be fully transformed into C source or LLVM assembly, which is compiled with

Table 1. Transactional primitives.

`XACT_BEGIN`	Begins a transaction.
`XACT_END`	Commits a transaction.
`XACT_PAUSE`	Pauses a transaction.
`XACT_UNPAUSE`	Resumes a transaction.
`XACT_ABORT`	Intentionally rolls back a transaction, optionally jumping to a specified location afterward.
`XACT_RETRY`	Suspends a transaction, restarting it when its read set is modified.
`PXACT_ADD`	Adds a memory location to a pseudotransaction.
`PXACT_DEL`	Removes a memory location from a pseudo-transaction.

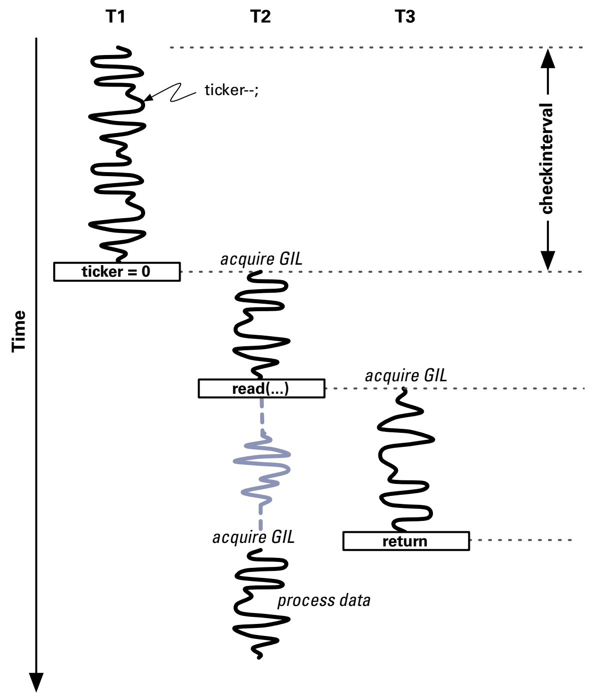


Figure 1. PyPy and CPython's Global Interpreter Lock (GIL)-based threading model. The `ticker` variable is decremented after each bytecode is executed.

the respective compiler and linked with hand-written C code to produce a functional interpreter.

Extension modules for PyPy are likewise implemented in a combination of RPython and C. However, PyPy is evolving a foreign function interface so that extension modules can be written in RPython alone, and compiled for either PyPy or CPython. Reducing the amount of native code in extension modules enables their implementations to be analyzed and transformed more thoroughly, which benefits transactional execution as we discuss in Section 3.6.

Fig. 1 depicts PyPy's threading model, which like CPython's permits at most one OS-level thread to execute Python code, by employing a lock known as the Global Interpreter Lock (GIL). A thread holds the GIL while interpreting a user-configurable (via `sys.setcheckinterval`) number of Python bytecode instructions, by default 100.[1] When the thread releases the GIL, another thread waiting on the lock can acquire it. Extension modules may release the GIL before performing system calls which are likely to block, thereby allowing other threads to execute Python code while they wait. Module code reacquires the GIL when the system call completes.

The use of the GIL ensures Python bytecodes are executed in isolation, relieving runtime implementers and extension module developers from having to reason about interactions between concurrently executing bytecodes. Hardware-supported transactions provide this same isolation, but do so without sacrificing parallel execution by optimistically executing bytecodes in parallel, detecting (and recovering) when they touch the same data.

Transactional Python execution requires—whether transactions emulate the semantics of lock-based concurrency, events and continuations, or directly execute new code written with explicit transaction boundaries—the ability to wrap arbitrary code executing within the Python interpreter in a transaction. To this end, we begin by enclosing the execution of each Python bytecode in a separate transaction.

An individual thread enables or disables per-bytecode transactions with the `sys.settrans` function. This function releases the GIL (which the thread was currently holding) and begins a transaction. The disabling process is similar: the current transaction commits and its thread reacquires the GIL. While per-bytecode transactions are enabled, at the point between bytecodes at which a thread would check its `ticker` counter to determine if it should release the GIL, the transaction commits and a new transaction begins.

Per-bytecode transactions are semantically equivalent to setting the GIL check interval to 1 (a single bytecode), assuming the GIL is not released by an extension module. While this change theoretically enables bytecode-granuarity concurrency, it is not sufficient to achieve significant parallelism due to sources of false conflicts—those introduced by the interpreter's implementation, rather than the executing Python code itself. The following section discusses these conflicts, and the manner in which we minimize them.

3.2 Eliminating Common False Conflicts

There are three primary sources of false conflicts in the PyPy interpreter: 1) all threads share a single copy of exception information, which is frequently written to and read, 2) a performance optimization where the running thread's stack base pointer is cached in a single global variable, and 3) conflicts resulting from dynamic resolution of symbols.

Each thread stack includes a pointer to an `ExecutionContext` object that encapsulates most thread-specific PyPy interpreter state, thereby ensuring that a thread is ready to execute once it acquires the GIL. However, PyPy does not maintain per-thread exception information, which presents an issue when multiple active threads execute Python code in parallel. Python exceptions are objects, and PyPy represents the current exception state as a global structure containing an (*exc_type*, *exc_value*) pair of global variables. The PyPy interpreter code or extension modules may raise an exception by setting these variables, and the exception type is checked following many interpreter function calls to determine if an exception handler should be invoked.

Given the frequency with which the exception state is accessed, concurrently executing Python threads experience many conflicting accesses to this structure. This is easily remedied by changing reads and writes of exception state to reference thread-local storage. Newer Linux kernels provide low-overhead access to thread-local storage by utilizing an IA-32 segment register, so the performance impact of this change is relatively small (see Section 4).

While each thread maintains a copy of its stack base pointer in thread-local storage, a copy of this pointer for the current running thread is cached in a global variable for performance. The frequently called `LL_stack_too_big` function uses this cached copy to determine if the stack has overflowed and update the cached copy when necessary. While we could simply remove the cached copy, we instead chose to detect stack overflows in a manner similar to the way in which virtual memory hardware detects accesses to invalid pages. We replace the function entirely by initially estimating the base pointer on each thread, and using a pseudo-transaction to check for potential overflows. The `SIG_PXACT` signal handler, invoked on a potentially stack-overflowing memory access on a thread, either updates the estimate or modifies the thread's exception state to indicate an actual overflow. In cases where the runtime manages allocation of its own thread stacks, we could also use the page-level protection provided by the virtual memory hardware to detect stack overflows; however, in embedding applications, where

[1] The GIL surrounds most, but not all, complete bytecode executions; an obvious example is a Python function invocation, where the bytecode does not actually complete until the function returns. Nevertheless, the GIL is released at times when a thread switch is safe.

the runtime's thread stacks may be shared with its host, the additional flexibility of protection through pseudo-transactions may be beneficial.

The last source of false conflicts we experienced was not related to PyPy itself, but to the behavior of the dynamic loader. In some cases, on the first access to a symbol—`errno` was a common culprit—its location is computed and stored in a table. Another transaction, attempting to access the same symbol, first reads the table to determine whether the address has been computed, and is aborted or stalled as it attempts to read from an address that another uncommitted transaction had written. We worked around this problem by ensuring `errno` was accessed before transactional execution began; a potential general solution, which we did not explore, would be to pause the current transaction while relocation occurs, thereby ensuring the symbol's location is immediately visible to future execution regardless of the transaction, and avoiding incorporation of linkage tables into the transaction footprint. In a HTM-aware OS, this modification could be performed by modifying the loader or C library—below the application level, such that a dynamic language implementer would not need to handle it.

We eliminate some false conflicts by selecting appropriate PyPy translation options. For example, if we had configured PyPy with a reference counting GC, as CPython uses (in PyPy, it results in worse performance), we would experience many false conflicts when accessing the reference counts of common constants such as small integers and `None`.

3.3 Lock-Transaction Coexistence

In unmodified CPython or PyPy, the GIL ensures atomic bytecode execution: because operations on built-in Python data structures execute within a single bytecode, they are guaranteed to be atomic. Python also allows user programs to use locks for mutual exclusion. In PyPy, the RPython implementation of a lock acquire is:

```
def descr_lock_acquire(self, space, waitflag=True):
    GIL = space.threadlocals.GIL
    GIL.release()
    result = self.lock.acquire(bool(waitflag))
    GIL.acquire(True)
    return space.newbool(result)
```

To avoid deadlock, a thread releases the GIL before trying to acquire a lock. Holding the GIL would prohibit other threads from executing, including a thread that held the requested lock, resulting in the lock never being released. The thread reacquires the GIL after acquiring the lock. If the lock is already held and `waitflag= False`, the function returns `False` immediately. (The underlying lock objects, such as `space.threadlocals.GIL` and `self.lock` above, are implemented with POSIX semaphores or their equivalent.)

Our goal in converting locks to transactions, is to guarantee the mutual exclusion denoted by the lock, while allowing concurrent execution of independent critical sections. Thus, we would like to avoid actually acquiring the lock whenever possible, instead relying on the transactional memory to provide isolation. Nevertheless, our code has to correctly handle all cases, *i.e.*, both GIL-holding and transactional threads. The RPython portion of the implementation is shown in Fig. 2. It is somewhat reminiscent of a multiple reader/single writer lock, in that any number of threads are allowed to simultaneously hold the lock within a transaction, but if the lock is actually acquired (typically by a GIL-holding thread) then no threads are allowed to hold it within a transaction.

To "acquire" a lock when per-bytecode (PBC) transactions are enabled, we commit the current PBC transaction and begin a transaction whose scope will correspond to the lock's (lines 3–5). First considering the case where `self.needs_GIL` is false, the code attempts to *reserve* the lock (line 10). Reserving a lock is a nonblocking operation which reads the lock's state, returning true if the lock is free. The process of reserving a lock adds it to the transaction's read set, which will ensure that if another thread acquires the lock, this thread's transaction will be aborted. It also records the lock in thread-local storage, so that an attempt to unreserve the lock later will be considered safe. If the lock is free, we begin a PBC transaction, nested inside the lock transaction (line 12). If the lock is not free and the lock "acquisition" attempt is non-blocking (*i.e.*, the `waitflag` is false), we end the lock transaction in order to release the read set, such that a true acquisition attempt will not cause an abort, and begin a PBC transaction (lines 16–17).

If the lock is not free and `waitflag` is true, the thread uses the transactional memory support to wait until the lock is released. The code (line 14) executes a *retry* operation [8], which stalls or deschedules the thread, but retains its transaction's read set. The thread is woken to rerun its transaction when any memory it touched has changed, usually indicating the lock has been released.

"Releasing" a lock in a PBC transaction begins by *unreserving* the lock (line 28), which checks that the lock being unreserved corresponds to a lock which has been acquired within the scope of the outermost transaction. We then perform the reverse of the transaction behavior on a successful "acquire", ending the PBC and lock transactions, and beginning a new PBC transaction (lines 29–31). The check on release is necessary to ensure that improper lock usage does not result in behavior that would not be permitted in the equivalent nontransactional execution. It can fail because the lock was originally acquired in nontransactional execution, or because the lock has already been released in transactional or nontransactional execution. Each of these conditions triggers an exception, aborting transactional execution.

3.4 Memory Allocation

Our HTM cannot undo all memory allocations without explicit support, because allocation can include requests to the operating system for additional address space. A transaction's scope does not expand to include operating system kernel code—while a transaction is overflowed, its footprint is tracked via virtual addresses. Instead, our HTM permits calls to the operating system only while no transaction is running on the current thread, or inside a transaction pause–a section of nontransactional code executed on the same thread as the containing transaction.

In the absence of garbage collection, care is required to avoid leaking memory when memory allocating transactions abort. Because memory allocation occurs in a transaction pause, allocated memory is immediately removed from the pool of free memory. As a result, an aborting transaction must explicitly undo its allocations to avoid leaking memory. In our system, non-garbage collected programs record each allocation as it occurs during the transaction and execute compensation code on a transaction abort to free that memory. We provide this functionality through an transaction-aware version of `malloc` and `free`, which programmers can use without having to worry about the effects of transaction aborts. A garbage collected environment makes this compensation code unnecessary.

PyPy currently defaults to using the Boehm-Demers-Weiser conservative garbage collector. Allocation is primarily thread-local, but periodically the collector needs to "stop the world," suspending all threads using the collector to ensure it has a consistent view of memory to examine. While it is possible to achieve performance improvements in a garbage collector by exploiting transactional isolation, we chose to ensure correct execution by modifying the "stop the world" behavior (implemented with POSIX signals) to abort any transactions in progress and delay transaction creation until the collection has completed nontransactionally. Without this

```
class Lock(Wrappable):
    def descr_lock_acquire(self, space, waitflag=True):
        if rtrans.is_active():
            rtrans.end() # end PBC transaction
            rtrans.begin() # begin lock transaction
            if self.needs_GIL:
                result = self.lock.acquire(bool(waitflag))
                space.getexecutioncontext().settrans(space.w_False) # end lock transaction, acquire GIL
            else:
                result = self.lock.reserve()
                if result:
                    rtrans.begin() # begin PBC transaction
                elif waitflag:
                    rtrans.retry()
                else:
                    rtrans.end() # end lock transaction
                    rtrans.begin() # begin PBC transaction
        else:
            GIL = space.threadlocals.GIL
            GIL.release()
            result = self.lock.acquire(bool(waitflag))
            GIL.acquire(True)
        return space.newbool(result)

    def descr_lock_release(self, space):
        try:
            if rtrans.is_active():
                self.lock.unreserve()
                rtrans.end() # end PBC transaction
                rtrans.end() # end lock transaction
                rtrans.begin() # begin PBC transaction
            else:
                self.lock.release()
                from pypy.rpython.objectmodel import we_are_translated
                if we_are_translated() and self.lock.lockedintrans():
                    space.getexecutioncontext().settrans(space.w_True)
        except thread.error:
            # [raise exception, lock already unlocked]
```

Figure 2. RPython modifications to PyPy's locking implementation, providing lock-transaction coexistence.

simplifying assumption, the garbage collector would need to be made aware of transactions to ensure it finds all the live objects, because during a transaction we might be keeping two versions of an object alive: one to be used if the transaction commits and one in case it aborts. In our experience, the garbage collector only "stops the world" approximately every 100,000 per-bytecode transactions: therefore, the performance impact of aborting a few transactions each time is minimal.

Extension module code may invoke libraries which do not use the PyPy garbage collector, and may attempt to perform system calls to allocate memory inside an unpaused transaction. Memory allocation is not the only such action extension modules can perform; this situation is detected and handled using a technique discussed in the following section.

3.5 Non-Undoable Actions

By wrapping native code execution in hardware-supported transactions, undoable operations can be executed in parallel, where an "undoable" operation is defined as one which can be reversed by undoing a thread's stores and restoring the thread's register checkpoint from the beginning of the transaction. Some operations, such as I/O, cannot always be executed in an undoable fashion.

We distinguish three classes of non-undoable operations.

1. Operations that are safe to execute in place, but within a transaction pause, with or without a compensating operation to be executed if the transaction is aborted. Several examples of these operations have already been discussed: memory allocation in the presence of garbage collection (requiring no compensation) or without garbage collection (requiring a compensating free), and symbol relocation (idempotent, requiring no compensating action).
2. Operations which are safe to defer until the completion of the outermost transaction. For a non-undoable operation to be safe to defer, the remainder of the transaction must not depend on its results. In future transactional memory-aware operating systems and libraries, deferral would be a standard feature of system calls.
3. Operations with dependencies later in the transaction, and which cannot be reversed with compensation.

When only a per-bytecode transaction is in progress—that is, the transaction nesting depth is 1—almost all non-undoable operations fall into class 1. Operations in class 3 may only be performed nontransactionally. As discussed in Section 3.6, it's possible to optimistically execute an operation as if it is in class 2, reverting to nontransactional execution (class 3) if deferral is not possible.

Our transactional memory implementation is intentionally designed to prohibit transactions from executing in the kernel. Unless the current transaction is paused, a system call within a transaction is considered an error, aborting the transaction. To execute operations of class 1 in a per-bytecode transaction, we could trap the abort, acquire the GIL and reexecute nontransactionally, but a more efficient implementation is possible: *automatic transactions*.

Automatic transactions are designed to mirror the conditions under which the GIL is released around system calls. The current transaction commits before entering the system call implementation, and a new transaction starts upon completion of the call. This is a safe transformation because system call implementations don't touch internal PyPy state. For existing extension modules, it's permissible to split the code before and after a system call into separate transactions, based on the manner in which the GIL is released around blocking I/O. If a conflict occurs in the second transaction, after the system call, there is no need (nor any ability) to reexecute the code before the system call: typically, this code unpacks the system call result into a data structure that can be manipulated from Python.

Binary rewriting or wrapper libraries could be used to trap system calls in user space before they enter the kernel. For simplicity's sake, we chose to implement automatic transactional behavior in the kernel. Enabling per-bytecode transactions in PyPy thus invokes a system call which sets the `TIF_SYSCALL_AUTO_XACT` flag in the current thread's kernel state. On a system call entry with automatic transactions enabled, instead of aborting the current transaction, it is committed. As the system call returns to user code, a new transaction is begun.

Automatic transactions can only be applied in per-bytecode transactions, therefore reserving the first lock in a thread disables automatic transactions; unreserving the last lock in a thread reenables them. When an undeferrable system call occurs in a lock-based transaction, it aborts. The runtime uses this abort as a signal that the region protected by the lock is not transaction-safe, and should be reexecuted after acquiring the GIL.[2]

Reserving the first lock in a thread also attaches abort compensation code to its transaction. The abort compensation code queries the HTM system to identify the type of abort performed. If the abort was performed by the system call handler, then in a transaction pause (so that the change can live beyond the aborted transaction) the `needs_GIL` attribute of the RPython lock object corresponding to the outermost transaction is set to `True`. The remainder of the code in Fig. 2, which we have not yet discussed, uses this attribute. If the lock needs the GIL, we acquire the lock and disable per-bytecode transactions (lines 6–8). Lock acquisition within the transaction (line 7) records the lock in a thread-local variable, so that per-bytecode transactions can be resumed when the lock is released (lines 35–36). In our current implementation, the `needs_GIL` attribute persists once it is set; more context-sensitive matching strategies may allow greater concurrency.

Non-undoable operations do not always need to be executed in place, if their results are not needed within the scope of the transaction. The next section discusses a method for deferring these operations to permit additional transactional execution.

3.6 Optimistic Deferral

Through standardized methods for type conversion and function invocation available in generic foreign function interfaces, it is possible to introspect the process of data exchange with native functions. In cases where automatic transactions cannot be used, pseudo-transactions watch the data returned by a system call-invoking foreign function, and we attempt to defer the function's execution until the end of a transaction, rather than falling back to nontransactional execution.

`ctypes` [10] is a foreign function interface for Python, included with the forthcoming CPython 2.5 release; `rctypes`, a RPython implementation, is the primary means of development for PyPy extension modules. It allows Python code to directly call C functions and manipulate arbitrary C data structures and types. As `ctypes` can invoke a native function and return its results to Python code, it must understand the correspondence between the arguments and return types of the function and the corresponding Python data types. For example, a `ctypes` declaration and invocation of the `strerror` function, with prototype `char *strerror(int errnum)`, looks like:

```
from ctypes import util, cdll, c_int, c_char_p
libc = cdll.LoadLibrary(util.find_library('c'))
strerror = libc.strerror
strerror.argtypes = [c_int]
strerror.restype = c_char_p
strerror(22) # returns 'Invalid argument'
```

Fig. 3 shows a simple example of deferral, involving a foreign function `f` with a single parameter and return value. In the first scenario, a modified `ctypes` module registers abort compensation code which, if the abort happens because a system call was attempted, sets the thread-local `ctypes_use_proxy` flag. On the second attempt, `ctypes` does not attempt to invoke the system call, but instead saves the converted Python argument `na`, registers compensation code to perform the invocation, and initializes the would-be return value with a proxy, whose target object is watched in a pseudo-transaction.[3]

For the remainder of the transaction in the second and third scenarios, the pseudo-transaction monitors use of the proxy's fake target. Manipulating the proxy `b` itself, as in the second scenario where it is added to a list, is acceptable. Any attempt to perform operations on the contents, however, will be directed through the pointer to the proxy's target and triggers a pseudo-transaction conflict handler which reverts to nontransactional execution, shown in the third scenario.

One case which we cannot easily address is a foreign function which both calls back into the Python runtime and invokes a system call. As the purpose of foreign function interfaces is to eliminate the necessity for language-specific wrappers, the chances of this occurring are small, but in pathological situations we may not be able to account for some information leakage.

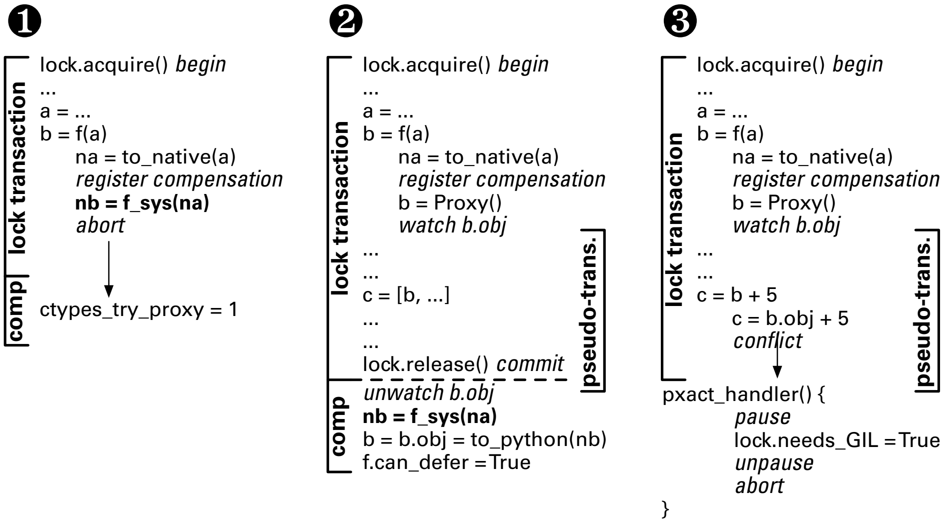
Because `rctypes`-based extension modules are still in development, we were unable to test the applicability of optimistic deferral.

3.7 Safety

Hardware-supported transactions are not only useful for concurrent execution: the checkpointing facilities they provide can be used with explicit aborts for error recovery. If native code misbehaves when invoked from within a per-bytecode transaction we may be able to restore the system to a consistent state by aborting the transaction. Such a resolution could succeed when a native function receives erroneous input from Python code, for example. In order for this scheme to work, we must detect the error within the scope of the transaction, and process the error, since simply reexecuting the transaction as in a typical abort most likely will cause the error to recur.

[2] The GIL is read at the beginning of each per-bytecode transaction. Acquiring the GIL thus aborts per-bytecode transactions in progress, serializing bytecode execution until the GIL is released.

[3] Not shown is a nested transaction, begun after the deferred call. If the system call result triggers an exception handler outside the lock transaction, we perform an explicit abort of the inner transaction, a commit of the lock transaction, and resume execution in the exception handler.



```
❶
lock.acquire() begin
...
a = ...
b = f(a)
    na = to_native(a)
    register compensation
    nb = f_sys(na)
    abort

ctypes_try_proxy = 1
```

```
❷
lock.acquire() begin
...
a = ...
b = f(a)
    na = to_native(a)
    register compensation
    b = Proxy()
    watch b.obj
...
...
c = [b, ...]
...
...
lock.release() commit
unwatch b.obj
nb = f_sys(na)
b = b.obj = to_python(nb)
f.can_defer =True
```

```
❸
lock.acquire() begin
...
a = ...
b = f(a)
    na = to_native(a)
    register compensation
    b = Proxy()
    watch b.obj
...
...
c = b + 5
    c = b.obj + 5
    conflict
pxact_handler() {
    pause
    lock.needs_GIL =True
    unpause
    abort
}
```

Figure 3. Optimistic deferral. 1) A system call aborts the transaction; abort compensation code marks the transaction for optimistic deferral. 2) Successful deferral; the system call is executed in the commit compensation code. 3) Unsuccessful deferral; transactional execution is not possible.

	pystone iterations/s	richards ms/iteration
pypy-c-thread (r27485)	9434	3791
+ thread-local exceptions	8881	3916
+ transactions	8403	4304

Table 2. Single-threaded performance.

We register signal handlers for a set of error-related signals which are delivered synchronously to the current thread: SIGSEGV, SIGILL, SIGFPE, SIGBUS, SIGSYS and SIGABRT. Unlike system calls and interrupt handlers, signal handlers are executed in user space, in the context of the current transaction. The signal handlers first test transactional execution is in progress. If so, they set the runtime's exception state to reflect the error in a paused region, so that it will be visible after the abort. Currently we provide only a description of the signal in the exception, but it would also be possible to provide any other information visible at the time of the error. After the signal handler executes an abort, execution resumes in the main bytecode interpretation loop, which tests the exception state before attempting to execute anything. PyPy dispatches the exception in the context of the bytecode to be executed.

This mechanism detects some of the most common failure conditions, such as an attempt to access an unmapped address or execute an illegal instruction. It is obviously inadequate if incorrect execution occurred prior to the beginning of the transaction: the error condition is then unrecoverable. In this case, the attempt to raise a Python exception may also fail. The signal handler thus first checks to see if an exception condition has been set on the current thread; if it has, it executes the preexisting behavior for that signal as in nontransactional execution (usually involving abrnomal termination of the process).

4. Evaluation

We begin our evaluation by considering the impact of our changes to the single-threaded performance of the PyPy interpreter. HTM primitives such as XACT_BEGIN are expressed by writing a primitive identifier and its parameters (if any) to registers, and executing a particular "magic" no-op instruction which our HTM simulator traps. Therefore, when code utilizing our HTM is run on unmodified hardware, the transactional primitives do nothing. Because single-threaded code never conflicts, we can run transactional PyPy on a real machine to approximate the single-threaded performance of our HTM if transactions never overflow into main memory—which is a reasonable assumption for per-bytecode transactions, as we discuss later in this section.

Table 2 includes results for two traditional Python benchmarks provided with PyPy: pystone is computationally intensive procedural code, and richards is object-oriented. The first row of results corresponds to "vanilla" pypy-c configured with threading support, with no changes. The second row corresponds to a version with thread-local exception information, which incurs a 6% performance penalty for pystone and a 3% penalty for richards: PyPy does not need to check the exception state after every operation, so the proportion of exception checks to code varies. The last row represents the same code we used in the simulated HTM environment, with modifications to the Boehm GC, main interpreter loop, GIL and other locks, for an overall 12–13% slowdown. We performed these tests on a single-processor AMD Athlon64 3500+ machine with 1 GB RAM running Fedora Core 3 Linux (kernel 2.6.10-1.760_FC3).

As our simulation environment does not model instruction latency and cache behavior, we cannot accurately measure the performance of multithreaded transactional execution. Instead, we examine the feasibility of executing PyPy with per-bytecode transactions and the potential for conflicts between threads. The small number of changes we have made to an existing Python runtime results in a conflict-free parallel execution, assuming the Python code being excuted has no conflicts.

To test our baseline transactional PyPy implementation, we profiled between one and four threads concurrently executing the pystone and richards benchmarks with per-bytecode transactions enabled in our HTM simulator. The version of richards presented here reduces the problem size to decrease the simulation's running time; this does not affect its qualitative behavior. An additional, monitoring thread is idle, waiting for the computational threads to complete.

	`pystone.1t`	`.2t`	`.3t`	`.4t`	`richards.1t`	`.2t`	`.3t`	`.4t`
Per-Transaction								
Reads	155.4	155.5	150.2	148.1	241.8	242.2	242.3	242.3
Bytes Read	598.0	598.1	577.2	569.2	939.0	940.4	940.7	941.0
Writes	95.7	95.6	92.0	90.7	151.5	151.5	151.6	151.6
Bytes Written	381.2	380.7	366.4	361.3	602.3	602.6	602.7	602.9
Instructions	494.3	470.7	466.6	455.3	743.8	739.2	739.8	740.5
Overall								
Begins	657	1302	1675	1814	56003	111975	167176	212374
Commits	657	1302	1675	1814	56003	111975	167176	212374
(Swapped)	4	5	9	15	1	19	45	96
Aborts	0	0	0	0	0	0	0	0
Swaps	117	112	126	145	1808	3431	5737	7264
In Progress	1.0	1.596	2.169	2.783	1.0	1.666	2.325	2.923

Table 3. Non-conflicting workload characterization.

Table 3 includes, for each execution, the following statistics:

- the average number of memory reads and writes performed and bytes read and written, per transaction (or partial transaction prior to an abort)
- the average number of instructions executed per transaction (not including paused sections or kernel code)
- transaction begins encountered (whether for the first time, following a retry, or an explicit, conflict, or system call-related abort)
- transactions successfully committed
- transaction swapped commits (an automatic in-kernel commit upon a system call, part of automatic transaction handling)
- transactions aborted (for any reason)
- how many times a transaction was swapped (entered kernel code on a context switch, system call or interrupt processing)
- the number of transactions in progress at the time a transaction begin was processed—an approximation of the achieved concurrency

Statistics collection begins when the last thread starts to execute the benchmark code and ends when first thread stops executing per-bytecode transactions; this is particularly noticeable in the overall statistics for `pystone`. The absence of transaction aborts in these executions demonstrates that we have identified and removed the common causes of false conflicts. The PyPy interpreter design presented few obstacles to per-bytecode transactional parallelization. The average memory footprint and length of per-bytecode transactions are well suited for in-cache management, which indicates they should execute efficiently, without the need for virtualization in proposed HTM systems.

Next, we examine the conflicts caused by a microbenchmark which repeatedly increments a global variable inside a transaction. Table 4 compares an interpreted PyPy version with a C compiled version. Because the body of the interpreted execution is significantly larger, we see fewer conflicts than in the compiled code. The number of aborts in the C version is not even larger because the overhead of managing the in-memory data structures representing transactions dominates the tiny transaction bodies.

	`incr-c.4t`	`incr-pypy.4t`
Per-Transaction		
Reads	1.0	159.9
Bytes Read	4.0	615.4
Writes	0.8	93.3
Bytes Written	3.1	372.1
Instructions	2.8	477.8
Overall		
Begins	10001	10001
Commits	7623	9946
(Swapped)	0	3
Aborts	2376	27
Swaps	2	156
In Progress	1.29	2.955

Table 4. Increment conflict characterization.

5. Related Work

One previous attempt has been made to implement a lightweight GIL-free Python runtime: Greg Stein's "free threaded" version of CPython 1.4, which locked individual Python data structures. This incurred a 30–50% performance penalty on a uniprocessor versus GIL-threaded code [25]. The implementation was not further pursued because of its performance penalty and limited applicability at the time (1996)—Microsoft Merchant Server 1.0 used a free threaded CPython, but subsequent versions were written in C++. However, this work resulted in CPython's current architecture for per-thread state storage [26].

Oplinger and Lam discuss the use of speculative parallelization to perform monitoring [19], applying automated instrumentation tools to detect buffer overflows and other misexecution. Our use of pseudo-transactions in monitoring for stack overflows, and performing transactional aborts when signals are received, involve much lower overheads but do not attempt such sophisticated program analysis.

Not all dynamic languages were designed from the start without consideration for parallel execution; for example, the latest revision of Erlang has incorporated multiprocessor support transparently to many applications [2].

The work we have described here sets the stage for user-exposed transactions in PyPy: it allows existing threaded and lock-synchronized code to execute concurrently where possible, reverting to GIL-threaded behavior if necessary. While our HTM implementation allows us to replace locks with carefully designed transactions that correctly interoperate with explicit locks, detect and handle attempted I/O, such optimistic lock-to-transaction conversion may be directly supported in hardware. In particular, Speculative Lock Elision (SLE) [21] enables concurrent execution of multiple threads holding the same lock as long as their behavior does not conflict. Speculative state is buffered in hardware, similar to the "local" or non-overflowed mode of VTM.

SLE could be directly applied to PyPy's GIL, though the interpreter would still need to be modified to eliminate false conflicts and ensure interpreter state is thread-local. As described, SLE does not support nested locks, and does not expose transactional primitives to user code, so it would not transactionalize Python-level locks, nor offer the programming model and safety benefits of true HTM. SLE would achieve approximately the same degree of concurrency and conflicts as described in the previous section, with lower overhead as lock reservation would be handled entirely in hardware. Single-threaded performance would be closest to that in the "thread-local exceptions" row of Table 2.

6. Conclusion

For parallel programming to become not only possible, but commonplace, programming languages and their implementations must provide a simple, practical model of concurrency. Transactions are such a model. Even when executing applications which do not explicitly employ transactional concurrency, the features provided by a hardware transactional memory infrastructure can offer dynamic language runtimes many benefits.

Transactional memory's properties of atomicity, isolation and composability, as enforced by the operating system and hardware, are uniquely suited to the challenges of evolving lightweight dynamic language runtimes, while maintaining the properties of simplicity and accessibility that have made these runtimes successful. Atomicity provides a highly flexible and easy-to-understand parallel programming model; isolation permits existing runtimes and their native code extensions to be easily adapted to safe parallel execution, and composability lets us convert locks to transactions at multiple levels, avoiding deadlock.

Our HTM implementation offers dynamic languages such as Python not only the ability to introduce a model of concurrency which is simple to use and implement, but the opportunity to simply improve safety and coexistence with arbitrary, preexisting native code. Our prototype demonstrates that transactional execution of Python is possible without disturbing the semantics of explicitly threaded Python code or compatibility with extension modules and embedding applications. Each mechanism we have presented dynamically reverts to nontransactional execution when necessary, to ensure compatibility and correctness. This showcases a significant benefit of HTM, in that transactional properties can be enforced across boundaries of implementation languages, runtimes and frameworks without requiring extensive mutual awareness of their interactions.

Acknowledgments

This research was supported in part by NSF CCR-0311340, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. We would like to thank Carl Friedrich Bolz, Michael Hudson, Samuele Pedroni and Armin Rigo for their assistance with PyPy. We thank Martin von Löwis and the anonymous reviewers for their valuable comments.

References

[1] Concurrency in Python. In *python-dev mailing list*, September 2005. URL `http://www.python.org/dev/summary/2005-09-16_2005-09-30.html#concurrency-in-python`.

[2] Erlang 5.5/OTP R11B highlights. URL `http://www.erlang.org/doc/doc-5.5/doc/highlights.html`.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[4] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[5] A. Bergman. Where wizards fear to tread: Perl 5.8 threads. URL `http://www.perl.com/pub/a/2002/06/11/threads.html`.

[6] C. Blundell, E. C. Lewis, and M. M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[7] F. Drake, Jr. et al. Thread state and the Global Interpreter Lock. In *Python/C API Reference Manual*. URL `http://docs.python.org/api/threads.html`.

[8] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[10] T. Heller. The `ctypes` module: an advanced foreign function interface for Python 2.3 and higher. URL `http://starship.python.net/crew/theller/ctypes/`.

[11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Symposium on Principles of Distributed Computing (PODC)*, 2003.

[13] B. Hindman and D. Grossman. Strong Atomicity for Java Without Virtual-Machine Support. Technical Report UW-CSE 2006-05-01, May 2006.

[14] T. Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, CA, March 2004.

[16] G. Lefkowitz and I. Shtull-Trauring. Network programming for the rest of us. In *USENIX Annual Technical Conference, FREENIX Track*, pages 77–89. USENIX, 2003. ISBN 1-931971-11-0.

[17] P. S. Magnussen et al. Simics: A full system simulation platform. *IEEE COMPUTER*, 35(2):50–58, February 2002.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[19] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Conference*

on Architectural Support for Programming Languages and Operating Systems, October 2002.

[20] PyPy. An implementation of Python in Python. URL `http://codespeak.net/pypy/`.

[21] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[23] A. Rigo. `py.magic.greenlet`: Lightweight concurrent programming. URL `http://codespeak.net/py/current/doc/greenlet.html`.

[24] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Dynamic Languages Symposium*, Portland, Oregon, October 2006.

[25] G. Stein. Population simulation in Stackless. In *Stackless mailing list*, August 2002. URL `http://www.stackless.com/pipermail/stackless/2002-August/000482.html`.

[26] G. Stein. Python at Google. In *PyCon*, 2005. URL `http://www.sauria.com/~twl/conferences/pycon2005/20050325/Python%20at%20Google.notes`.

[27] D. Sugalski and A. Bergman. `perlthrtut`—tutorial on threads in Perl. URL `http://perldoc.perl.org/perlthrtut.html`.

[28] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.

[29] D. Thomas and A. Hunt. Threads and processes. In *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, 2001. URL `http://www.rubycentral.com/book/tut_threads.html`.

[30] C. Tismer. Stackless Python: Tasklets. URL `http://www.stackless.com/wiki/Tasklets`.

[31] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, June 2006.

[32] C. Zilles and D. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, pages 48–55, June 2005.