Adding Concurrency in Python Using a Commercial Processor's Hardware Transactional Memory Support

Fuad Tabba
University of Auckland
fuad@cs.auckland.ac.nz

Abstract

This paper reports on our experiences of using a commercial processor's best-effort hardware transactional memory to improve concurrency in CPython, the reference Python implementation. CPython protects its data structures using a single global lock, which inhibits parallelism when running multiple threads.

We modified the CPython interpreter to use best-effort hardware transactions available in Sun's Rock processor, and fall back on the single global lock when unable to commit in hardware. The modifications were minimal; however, we had to restructure some of CPython's shared data structures to handle false conflicts arising from CPython's management of the shared data. Our results show that the modified CPython interpreter can run small, simple, workloads and scale almost linearly, while improving the concurrency of more complex workloads.

1 Introduction

The age of parallel computing is finally here. Most processor manufacturers today are producing chips with an increasing number of cores [8], while many software applications are yet to be able to take advantage of the concurrency these additional cores afford.

Transactional memory [9] is one proposal that promises to make it easier to reason about parallel programs, and by extension, to take advantage of the additional cores. Transactional memory is a programming model that provides a level of abstraction on top of critical sections by using transactions, which are "a sequence of actions that appears in-divisible and instantaneous to an outside observer" [11]. Transactions promise to make it easier to reason about each critical section individually, without being concerned with the possible interactions between them.

Different methods have been proposed for supporting transactional memory. Of interest to this paper is *best-effort* hardware transactional memory support, which may not guarantee all transactions to commit [12]. To date, best-effort support is the only type of hardware transactional memory support available on a commercial processor, such as Sun's Rock processor [4].

This paper reports on our experiences of using Rock's best-effort hardware transactional memory to improve concurrency in Python [18]. Python is a high level programming language with a design philosophy that emphasizes code readability. Since its release in 1991, Python has become one of the most popular programming languages[1], and is now a standard component of many operating systems such as Apple's OS X, Sun's Solaris, and various Linux distributions.

There are many different implementations of Python; the reference implementation is *CPython* [18], a byte-code interpreter written in C. CPython was developed before the multicore era, and therefore not designed with parallelism in mind. CPython supports multiple threading; however, CPython protects its critical sections using a single global lock, known as the *Global Interpreter Lock* (GIL) [19]. The GIL protects *all* accesses to CPython's data structures; therefore, the GIL serial-

---

[1]According to the TIOBE Programming Community Index for May 2010 [1], Python ranks as the seventh most popular programming language.
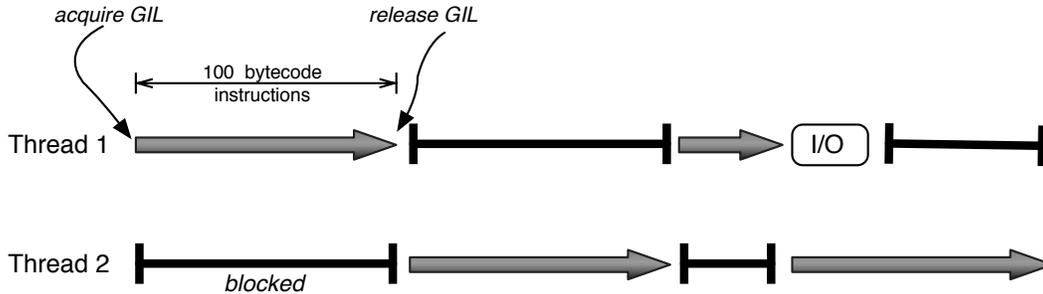
Figure 1: *An example of running two concurrent threads in CPython*

izes all parallel threads that access CPython's data structures. In practice, multiple threads can run in parallel only when executing external modules that do not acquire the GIL, and when performing a blocking operation such as I/O, in which case the thread releases the GIL before the operation begins, and attempts to reacquire the GIL once the operation is over.

A thread typically acquires the GIL to execute a number of byte-code instructions. This number is an adjustable runtime parameter, and can be as small as a single instruction. Because there is overhead involved in acquiring and releasing the GIL, the longer a thread holds the GIL the more it can amortize this overhead, which is why this number defaults to one hundred byte-code instructions. Figure 1 shows an example of two concurrent threads in CPython.

CPython creates and starts using the GIL only after the interpreter spawns more than one thread. Therefore, a single thread does not incur any locking overhead.

Neither the GIL, nor any aspect of its implementation, are part of the Python programming language specification. The GIL is an implementation detail used in the language's reference implementation, CPython, and others such as *PyPy* [15]. For example, *Jython* [10], a Python implementation that runs on the Java Virtual Machine, does not use global locks, but relies on the Java Virtual Machine's native concurrency mechanisms instead.

The GIL has often been a contentious issue within the Python community, with many criticisms of the Python language stemming from its use of the GIL [2]. Expert CPython developers tried to decompose the GIL into multiple fine-grained

locks; however, the resulting overhead in the single-threaded case — which is the common case for most Python programs — was a slowdown of a factor of two; so the attempt was abandoned [17].

This paper investigates whether Rock's transactional support could improve concurrency in CPython. The CPython interpreter was modified to use best-effort hardware transactions, and fall back on the GIL when unable to commit in hardware. The modifications were minimal; however, some of CPython's shared data structures were altered to handle false conflicts arising from CPython's management of the shared data. The modified CPython interpreter can run small, simple, workloads and scale almost linearly, while improving the concurrency of more complex workloads.

The main contributions presented in this paper are the following.

- We describe the changes to make CPython more concurrent and scalable, as well as the obstacles encountered in the process.
- We present a preliminary performance evaluation of the modified CPython interpreter on Rock. Other research groups have investigated using transactions to improve Python's performance [16, 3]; to the best of our knowledge, this work is the first to evaluate using transactions with Python on a real machine, rather than on a simulator.

The remainder of this paper is organized as follows. Section 2 describes the modifications to the CPython interpreter and the challenges encountered. Section 3 presents a preliminary performance evaluation of the concurrent CPython on Rock. In light of the evaluation, Section 4 discusses some of the possible design alternatives. Section 5 dis-

cusses some of the related work in using transactional memory to improve concurrency in Python. Finally, Section 6 concludes this paper.

2   Concurrent CPython

The target implementation was CPython 2.6.4, the most recent stable version at the time of the evaluation.

Because a best-effort hardware transactional memory is used, our approach was to apply lock elision to the GIL [13, 5]. The modified CPython first attempts to run a GIL critical section in hardware, and acquires the GIL only if the hardware transaction repeatedly aborts.

The granularity of the GIL was reduced to one byte-code instruction instead of the default one hundred, because Rock's best-effort hardware is not likely to be able to commit large transactions. This change was hardcoded to remove the overhead of checking the number of byte-code instructions the current critical section has run so far.

The code for acquiring and releasing the GIL was replaced with code that attempts the transaction in hardware, and falls back on the GIL if it repeatedly aborts. The hardware transaction first checks that the GIL is not acquired, and then proceeds with the critical section. This check is necessary to detect conflicts with other threads that acquire the GIL; if another thread acquires the GIL, this would immediately abort concurrently running hardware transactions.

When a hardware transaction aborts, it either tries the transaction again or acquires the GIL. The policy for deciding whether to fall back on the GIL when a hardware transaction aborts is based on the recommendations of Dice et al. [6], and can be summarized as follows.

If the transaction aborts because of an event where trying a transaction again is not likely to succeed in committing it, then the transaction immediately falls back on the GIL. These events include running an instruction Rock does not support inside a transaction (such as the divide instruction), and exceeding Rock's hardware limitations, such as its write buffer or cache memory size. If the transaction aborts because of an event where trying the

transaction again might succeed, then the transaction tries again a few more times, before falling back on the GIL. These events include coherence conflicts, mis-speculation, and TLB misses.

CPython releases the GIL when performing operations such as I/O, or when running external modules that do not access CPython's internal data structures. Because many of these operations are difficult to handle in hardware transactions [11], this eases the process of using transactions in CPython, and makes it more likely they will commit successfully in hardware.

When running in single-threaded mode, CPython does not check or acquire the GIL, because the overhead of acquiring the GIL is not necessary. The modified CPython maintains this aspect: when running in single-threaded mode, the modified CPython does not run any hardware transactions. CPython enters multi-threaded mode when the first additional thread is spawned, which initializes and acquires the GIL. There is no mechanism in CPython to exit multi-threaded mode, even after all spawned threads are destroyed.

Eliding the GIL was not by itself enough to make CPython run concurrently. The first tests just after eliding the GIL resulted in no additional scalability or concurrency. Most transactions were aborting because of conflicts with other transactions, even though the Python code for the tests does not logically share any data.

The cause of this problem was conflicts over global data structures, such as the ones used for memory management and for maintaining the current thread's state. Because CPython was not designed for concurrency, it defines many variables as global variables when they are conceptually thread-local. We annotated these variables with the `__thread` keyword to make them thread-local.

Not all global variables can be marked as thread-local without breaking some of the CPython invariants. For example, take the object `_Py_TrueStruct`, which designates the value representing `True`. This object is shared among all threads. Because CPython knows that there is only one instance of this object in the system, it often uses the value of *pointers* to this object to test whether a certain object is `True`, rather than dereference the pointer to check the actual

value it contains. If _Py_TrueStruct becomes a thread-local object, then all code that uses this shortcut must also be modified. This is because changing the _Py_TrueStruct into a thread-local object creates multiple instances of this object, all of which are a `True` object. Therefore, checking if an object is `True` just by checking whether it points to _Py_TrueStruct is not sufficient any longer. The same problem applies to other CPython objects, such as the Python *singletons* `False` and `None`.

Another difficulty in parallelizing CPython is its use of reference counting for memory management. With reference counting, every time an object is referenced a reference counter is incremented. When an object is no longer referenced, the counter is decremented. When the reference counter reaches zero, CPython knows that the object is not needed any more, and deallocates it.

When used with transactions, the problem with reference counting is that every transaction that accesses an object also modifies its reference counter. Transactions that access an object, even just for reading, modify the object. If the object is shared, then transactions that conceptually have no conflicts are conflicting, as far as the underlying transactional memory system is concerned.

This problem is exacerbated by false sharing, at the cache line level, between CPython objects. Some objects, such as the Python `True` and `False` objects, are sometimes allocated on the same cache line. Because Rock's hardware transactional memory detects conflicts at cache line granularity, a change to one object's reference counter is construed as a conflict with the other.

We solved the problem by creating a new `DoNotDeallocate` flag associated with every Python object. The flag is set for objects that exist for the whole lifetime of a CPython runtime instance. This flag indicates that there is no need to dynamically deallocate these objects and no need to track or update their reference counter. This eliminates false conflicts when accessing these objects in a transaction. It also obviates the need for declaring Python singletons, such as `True`, as thread-local variables. Instead, all singletons are flagged as objects that should not be deallocated.

Adding this flag could incur additional overhead, because every time a thread accesses an object, it must first check its `DoNotDeallocate` flag before deciding whether to adjust the object's reference counter. In practice, such checks can be eliminated, as well as all calls to try to increment or decrement an object's reference counter, in code where it can be determined that the object is one of these lifetime objects.

This solution is not suitable for all shared CPython objects, only for objects that have the lifetime of the interpreter. CPython also creates temporary shared objects, which need to be deallocated to avoid memory leaks. For example, CPython dynamically creates objects that represent Python code, functions, metadata such as variable names, and constants used in Python functions. We applied the same solution of using the `DoNotDeallocate` flag to these objects, even though it results in a memory leak. In our experiments, the memory leak was small, allowing us to proceed with the evaluation. Section 4 discusses possible solutions to this problem.

3 Evaluation

We evaluated the modified CPython on a pre-production prototype of Sun's Rock chip, with 16 cores in *SSE* mode, configured to run at 1.5 GHz; see Dice et al. [5] for details, where the prototype we use is referred to as "R2".

In Rock, function calls that use the `restore` instruction to pop the register window, when returning from the call, could cause transactions to abort [6]. To increase the likelihood of transactions committing in the presence of function calls, we instrumented `restore` instructions in the modified CPython interpreter using the `brnr` (branch if register not ready) instruction [6]. CPython was compiled using GCC 3.4 with optimization set to level 3.

We used two simple benchmarks that repeatedly modify a local variable in a loop: `iterate` and `count` [2]. Although both benchmarks accomplish the same general goal, each uses a different set of CPython byte-code instructions. The code listing below is for these two benchmarks.

4

```
def iterate(iterations):
    for x in xrange(0, iterations):
        pass

def count(iterations):
    while iterations > 0:
        iterations -= 1
```

These simple benchmarks are not representative of realistic workloads. We use them as a litmus test for whether CPython could run concurrently, and to evaluate eliding the GIL for a few simple CPython byte-code instructions.

For a more complex benchmark, we used `pystone`, a synthetic benchmark included in the CPython distribution. The `pystone` benchmark is a translation of the Dhrystone benchmark [22], a computationally-intensive integer benchmark.

When multiple threads are spawned, each CPython thread runs its own instance of these benchmarks. The `iterate` and `count` benchmarks perform 2,000,000 iterations each, and the `pystone` benchmark performs 10,000 iterations. The number of hardware transaction attempts is set to four, before falling back on the GIL.

We first evaluate the effect of the modifications on the single-threaded case, relative to the unmodified CPython. Before any threads are spawned, the modifications do not incur any overhead. CPython does not check or attempt to acquire the GIL in the single-threaded case, and the modifications do not attempt to run the code using hardware transactions in the single-threaded case either. The other modifications do not incur any overhead in the single-threaded case.

To observe the overhead the modifications incur on a single thread once the interpreter goes into multithreaded mode and starts using transactions, the benchmarks force CPython to go into the multithreaded mode by spawning and immediately destroying a thread. Table 1 presents the slowdown the modifications incur when using hardware transactions on a single thread, compared with running a single unmodified CPython thread, which incurs neither GIL nor transactional overhead.

The modified CPython incurs a slowdown of about a factor of three. This slowdown is mainly because Rock hardware transactional memory instructions and the code that attempts to elide the

Table 1: *Concurrent CPython slowdown relative to a single unmodified CPython thread*

Benchmark	Slowdown
iterate	2.8
count	3.9
pystone	2.1
average	2.9

GIL add significant overhead. Although this overhead is high, the modified CPython incurs this overhead *only* when running multiple threads, whereas the original implementation would *serialize* those threads.

Figure 2 presents the results for the performance and scalability going from 1 to 16 threads, relative to running a single unmodified CPython thread, which does not incur any synchronization overhead. These tests force the modified CPython to go into the multithreaded mode, by spawning and immediately destroying a thread, even when running only 1 thread.

Because these benchmarks do not conceptually share any data, ideally, they would scale linearly with the number of threads. Rock's hardware support is best-effort; if a transaction aborts, it acquires the GIL, which serializes other threads. Although we expect to see improvement in performance, we do not expect it to be linear with the number of threads.
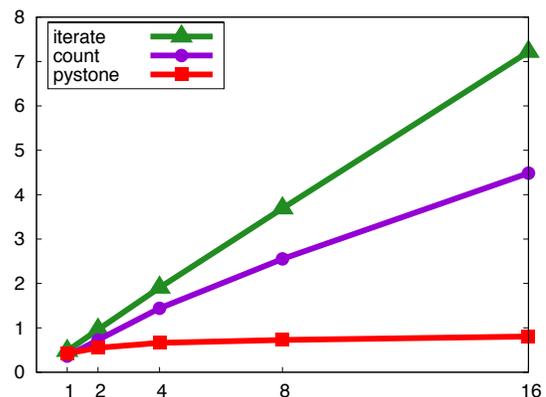

Figure 2: *Results of running the benchmarks using the modified CPython on the Rock machine. The x-axis represents the number of threads, the y-axis represents the speedup relative to a single unmodified CPython thread that does not acquire the GIL.*

5

The two simple benchmarks, `iterate` and `count`, scale well. Most of their hardware transactions commit, and from three threads onwards, they perform better than the unmodified single-threaded CPython. This is consistent with the initial observation that the modified CPython incurs a slowdown of a factor of three in the single-threaded case. We note that `iterate` scales better than `count`. The `iterate` benchmark uses CPython's built-in `xrange` function. The `xrange` function is optimized to be fast, and has a smaller memory footprint [21]; therefore, transactions involving that function are more likely to commit successfully on their first attempt.

The `pystone` benchmark is able to take advantage of some of the parallelism, but its performance does not improve much beyond four threads. The reason this benchmark does not perform as well as the other two is the high number of aborted transactions: about half of its transactions repeatedly abort and eventually fall back on the GIL. Those transactions are aborting because of mis-speculation, TLB misses, and because of function calls in transactions. To reduce the number of aborts caused by mis-speculation, we tried to instrument load instructions as well with a `brnr` speculation barrier [6]; however, the additional overhead of the barrier hurt performance more than it helped.

Finally, even though all `restore` instructions are instrumented with a `brnr` speculation barrier, the modified CPython still experienced aborts caused by function calls. We were unable to explain why the speculation barrier was not preventing these aborts; however, the nature of Rock's hardware support means that no transaction is guaranteed to commit, regardless of the safeguards.

Applying Amdahl's law, if half of the transactions fall back on the GIL, then the best speedup to expect is a factor of 2, assuming an infinite number of processors. This is consistent with our findings: the relative performance of `pystone` running on 16 threads to 1 thread is about 1.9.

These experiments, and Amdahl's law, highlight the potential problems of falling back on a mutually exclusive solution when transactions abort. As long as most hardware transactions commit, and the serial component remains small, then lock elision works well. However, even a small serial component has a big impact on scalability. A big serial component, such as the one in the `pystone` experiments, is devastating to scalability.

These experiments also highlight that, with little modification and some hardware support, existing programs can benefit immensely from transactional support — even from limited best-effort support such as the one in Rock. We expect that the modified CPython would improve as its underlying hardware support improves, without additional modifications to the software.

4   Design Alternatives

The main challenge in parallelizing CPython was in handling the operations that are treated as conflicts by the underlying system, even though they are conceptually nonconflicting. The two main causes of these problems in CPython are its use of global variables instead of thread-local variables, and its use of reference counting for memory management.

The solution to the problem of global variables did not require much effort. The solution we use instructs the compiler to create a thread-local copy of these global variables, which we believe is a satisfactory solution. On the other hand, the memory management problem is a more complex one.

CPython uses reference counting, instead of other methods of garbage collection, because it is more portable than the alternatives [20]. One alternative is to use the *Boehm-Demers-Weiser* conservative garbage collector, which would solve the problem reference counting causes [16]. Although portable, the Boehm-Demers-Weiser garbage collector does not run on all systems CPython supports.

The reference counting implementation CPython uses does not scale because CPython was not designed with scalability in mind to begin with. However, many research groups have proposed different techniques of making reference counting scalable. One possibility is to use the *Scalable Non-Zero Indicator* (SNZI), proposed by Ellen et al. [7], to represent reference counts. SNZI uses a hierarchal algorithm, and is typically implemented as a tree data structure distributed across several memory locations. Ellen et al. show that, because of

SNZI's hierarchical nature, it scales well when used as a reference counter, and works particularly well in the context of transactional memory.

Another alternative is to modify the underlying hardware so that it is aware of dependences and can forward the values of modified data between transactions, data such as modified reference counts. Ramadan et al. [14] propose a dependence-aware hardware transactional memory, which could mitigate the reference counting problem. Blundell et al. [3] have encountered the same problem and also propose a hardware solution, which is discussed in the next section. However, such support requires extensive changes to the underlying hardware, and does not solve the problem for existing systems.

In our opinion, given the resources available, the best solution to this problem is to use the Boehm-Demers-Weiser garbage collector. We did not investigate this solution, primarily because it was unclear if continued access to the Rock machine would be possible.

Another obstacle in the scalability of the modified CPython is caused by using the GIL as a fall-back mechanism. As long as most transactions commit in hardware, the GIL is successfully elided, and the system scales. As the abort rate increases, the GIL becomes a serious bottleneck. Using a hybrid solution that falls back on a software transactional memory could mitigate this bottleneck [12], because most software transactional memory proposals allow software transactions to run concurrently. It is important, however, that the software component does not introduce much overhead, particularly in the single-threaded case. Otherwise, the software transactional memory's overhead might negate most of the benefits from parallelism.

5 Related Work

Riley and Zilles [16] proposed substituting GIL critical sections with hardware transactions. Instead of CPython, they targeted the PyPy Python implementation, which supports using the Boehm-Demers-Weisser garbage collector, thus eliminating conflicts caused by updating reference counts.

Riley and Zilles evaluated their proposal on a simulated unbounded hardware transactional memory that does not model instruction latency and cache behavior, and therefore does not accurately measure performance. Their goal was to examine the feasibility of using transactions as a substitute for the GIL. For the workloads they evaluate, all transactions commit successfully. Based on the average memory footprint of their transactions, they concluded that best-effort hardware support is sufficient: the average number of bytes read in the biggest benchmark they used was less than 1 KiB, and the average number of bytes written by that benchmark was less than 640 bytes.

Rock's transactional write buffer holds only 32 entries. Therefore, assuming a 64 bit write buffer entry, the write buffer would be able to hold a maximum of 256 bytes, which is not sufficient for most transactions to commit in hardware for the workloads Riley and Zilles evaluated.

Concurrently with and independently of our work, Blundell, Raghavan, and Martin [3] proposed a transactional hardware mechanism that symbolically tracks modifications and constraints that a transaction applies to variables, and transparently applies the modifications and checks if the constraints still hold before a transaction commits. For example, if a transaction increments a reference counter, then the system applies the increment operation at commit time, regardless of the current value of the reference counter. Moreover, if a transaction checks that the value of a reference counter is greater than zero, then the transaction can commit as long as that constraint holds at commit time, i.e., the reference counter is still greater than zero.

The work of Blundell et al. targets conflicts on auxiliary data, which often cause bottlenecks in transactions with otherwise nonconflicting operations — exactly the type of conflicts CPython's reference counting creates. Blundell et al. modify CPython to use transactions to protect its critical sections, and present a simulator evaluation of applying their hardware scheme to the modified CPython. Their results show that, with their proposed modifications, CPython can scale almost linearly, at least up to 32 threads.

6 Concluding Remarks

This paper demonstrates that, with little modification, programs not designed for concurrency can leverage best-effort hardware transactional memory support to scale as the number of available cores increases. Towards that end, this paper presented an evaluation of using hardware transactions in CPython on a real machine that supports hardware transactional memory. Although the results are not conclusive, they demonstrate the potential even limited best-effort hardware transactional memory has for improving performance. We did not investigate this topic further or pursue other directions, primarily because it was unclear if continued access to the Rock machine would be possible.

Acknowledgements

References

[1] TIOBE programming community index for May. TIOBE Software BV, 2010.

[2] D. Beazley. Inside the python GIL (slides). Presented at the Python Concurrency Workshop, 2009.

[3] C. Blundell, A. Raghavan, and M. M. K. Martin. RET-CON: Transactional repair without replay. In *the proceedings of the 37th annual international symposium on Computer architecture*. ACM, 2010.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *IEEE Micro*, 2009.

[5] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *the proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. ACM, 2009.

[6] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. TR-2009-180, Sun Microsystems Laboratories, 2009.

[7] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable nonzero indicators. In *the proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007.

[8] D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 2005.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *the proceedings of the 20th annual international symposium on Computer architecture*. ACM, 1993.

[10] J. Juneau, J. Baker, F. Wierzbicki, L. S. Munoz, and V. Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, 2010.

[11] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.

[12] M. Moir. Hybrid transactional memory. Sun Microsystems Laboratories, 2005.

[13] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *the proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001.

[14] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *the proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008.

[15] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *the companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

[16] N. Riley and C. Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *the companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

[17] G. Stein. Free threading. Python Software Foundation, 2001.
http://mail.python.org/pipermail/python-dev/2001-August/017099.html.

[18] G. van Rossum. *The Python Language Reference: Release 2.6.4*. Python Software Foundation, 2009.

[19] G. van Rossum. *The Python/C API: Release 2.6.4*. Python Software Foundation, 2009.

[20] G. van Rossum. *Extending and Embedding Python: Release 2.6.4*. Python Software Foundation, 2009.

[21] G. van Rossum. *The Python Standard Library: Release 2.6.4*. Python Software Foundation, 2009.

[22] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 1984.