A Way Forward in Parallelising Dynamic Languages

Position Paper, ICOOOLPS'14

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

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

Abstract

Dynamic languages became very popular in recent years. At some point, the need for concurrency arose, and many of them made the choice to use a single global interpreter lock (GIL) to synchronise the interpreter in a multithreading scenario. This choice however makes it impossible to actually run code in parallel.

Here we want to compare different approaches to replacing the GIL with a technology that allows parallel execution. We look at fine-grained locking, shared-nothing, and transactional memory (TM) approaches. We argue that software-based TM systems are the most promising, especially since they also enable the introduction of atomic blocks as a better synchronisation mechanism in the language.

Keywords transactional memory, dynamic languages, parallelism, global interpreter lock

1. Introduction

In a world where computers get more and more cores and single-thread performance increases less and less every year, many dynamic languages have a problem. While there is certainly a lot of popularity around languages like Python and Ruby, their ability to make use of multiple cores is somewhat limited. For ease of implementation they chose to use a single, global interpreter lock (GIL) to synchronise the execution of code in multiple threads. While this is a straight-forward way to eliminate synchronisation issues in the interpreter, it prevents parallel execution. Code executed in multiple threads will be serialised over this GIL so that only one thread can execute at a time.

There exist several solutions and workarounds to remove or avoid the GIL in order to benefit from multiple cores. We are going to discuss several of them and try to find the best way forward. The first approach uses fine-grained locking to replace the single GIL. Then there are shared-nothing models that use for example multiple processes with multiple interpreters and explicit message passing. Finally, one can also directly replace the GIL with transactional memory (TM), either software-based (STM) or hardware-based (HTM).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICOOOLPS workshop 2014, July 28th, 2014, Uppsala, Sweden.
Copyright © 2014 ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The approach that wins in the end should perform similarly for single-threaded execution as compared to the GIL and be able to execute code in parallel on multiple cores. Furthermore, we will also take into account the compatibility to existing code that already uses threads for concurrency, as well as the changes that are required to the interpreter itself.

These requirements are not easy to meet. We argue that STM is the overall winner. While it has a big performance problem currently, it gets more points in all the other categories. We think that it is the only solution that also provides a better synchronisation mechanism to the application in the form of paralleliseable atomic blocks.

2. Discussion

In this section we examine the approaches and highlight their advantages and disadvantages.

2.1 Why is there a GIL?

The GIL is a very simple synchronisation mechanism for supporting multithreading in an interpreter. The basic guarantee is that the GIL may only be released in between bytecode instructions. The interpreter can thus rely on complete isolation and atomicity of these instructions. Additionally, it provides the application with a sequential consistency model [11]. As a consequence, applications can rely on certain operations to be atomic and that they will always be executed in the order in which they appear in the code. While depending on this may not always be a good idea, it is done in practice. A GIL-replacement should therefore uphold these guarantees, while preferably also be as easily implementable as a GIL for the interpreter. The latter can be especially important since many of these languages are developed and maintained by very large open-source communities, which are not easy to coordinate.

The GIL also allows for easy integration with external C libraries that may not be thread-safe. For the duration of the calls, we simply do not release the GIL. External libraries that are explicitly thread-safe can voluntarily release the GIL themselves in order to still provide some parallelism. This is done for example for potentially long I/O operations. Consequently, I/O-bound, multithreaded applications can actually parallelise to some degree. Again, a potential solution should be able to integrate with external libraries with similar ease. We will however focus our argumentation more on running code in the interpreted language in parallel, not the external C calls.

Since the GIL is mostly an implementation detail of the interpreter, it is not exposed to the application running on top of it. To synchronise memory accesses in applications using threads, the state-of-the-art still means explicit locking everywhere. It is known that using locks for synchronisation can be hard at times [12, 13, 16]. They are non-composable, have overhead, may deadlock,

	GIL	Fine-grained locking	Shared-nothing	HTM	STM
Performance (single threaded)	++	+	++	++	--
Performance (multithreaded)	--	+	+	+	+
Existing applications	++	++	--	++	++
Better synchronisation	o	o	o	o	++
Implementation	++	-	++	++	++
External libraries	++	++	++	++	++

Table 1. Comparison between the approaches (--/-/o/+/++)

limit scalability, and add to the overall complexity of the program logic. For a better parallel programming model for dynamic languages, we propose another, well-known synchronisation mechanism called *atomic blocks* [14, 15].

Atomic blocks are composable, deadlock-free, higher-level and expose useful atomicity and isolation guarantees to the application for a series of instructions. Interpreters using a GIL can simply guarantee that the GIL is not released during the execution of the atomic block. Of course, this still means that no two atomic blocks can execute in parallel or even concurrently. Potential solutions are preferable if they provide a good way to implement atomic blocks that are also able to be executed in parallel.

2.2 Potential Solutions

For the discussion we define a set of criteria to evaluate the multiple potential solutions for removing or avoiding the GIL and its limitations:

Performance: How well does the approach perform compared to the GIL on a single and on multiple threads?

Existing applications: How big are the changes required to integrate with and parallelise existing applications?

Better synchronisation: Does the approach enable better, paralleliseable synchronisation mechanisms for applications (e.g. atomic blocks)?

Implementation: How difficult is it to implement the approach in the interpreter?

External libraries: Does the approach allow for easy integration of external libraries?

2.2.1 Fine-Grained Locking

The first obvious candidate to replace the GIL is to use multiple locks instead of a single global lock. By refining the granularity of the locking approach, we gain the ability to run code that does not access the same objects in parallel. What we lose instead is the simplicity of the GIL approach. With every additional lock, the likeliness of deadlocks grows, as well as the overhead that acquiring and releasing locks produces. The former means that sometimes it is necessary to fall back to less fine-grained locking, preventing some potential parallelism in order to keep the complexity manageable. The latter means that we lose a bit of performance in the single-threaded case compared to the GIL, which requires much less acquire-release operations.

Jython [1] is one project that implements an interpreter for Python on the JVM[1] and that uses fine-grained locking to correctly synchronise the interpreter. For a language like Python, one needs quite a few, carefully placed locks. Since there is no central location, the complexity of the implementation is quite a bit greater compared to using a GIL. Integrating external, non-thread-safe libraries should however be very simple too. One could simply use one lock per library to avoid this issue.

[1] Java Virtual Machine

In the end, fine-grained locking can transparently replace the GIL and therefore parallelise existing applications, generally without any changes[2]. An implementation has to follow the GIL semantics very closely, otherwise it may expose some latent data races in existing applications which are just not exposed with a GIL. This approach does however not provide a better parallelising synchronisation mechanism to the application like e.g. atomic blocks.

2.2.2 Shared-Nothing

There are also approaches that work around the GIL instead of trying to replace it. If an application can be split into completely independent parts that only very rarely need to share anything, or only do so via an external program like a database, then it is sensible to have one GIL per independent part. As an example of such an approach we look at the *multiprocessing*[3] module of Python. In essence, it uses process-forking to provide the application with multiple interpreters that can run in parallel. Communication is then done explicitly through pipes.

Obviously not every application fits well into this model and its applicability is therefore quite limited. Performance is good as long as the application does not need to communicate a lot, because inter-process communication is relatively expensive. Also the implementation of this approach is very cheap since one can actually take an unmodified GIL-supported interpreter and run multiple of them in parallel. That way, we also inherit the easy integration of external libraries without any changes. While the model of explicit communication is often seen as a superior way to synchronise concurrent applications because of its explicitness, it does not actually introduce a better synchronisation mechanism for applications.

2.2.3 Transactional Memory

Transactional memory (TM) can be used as a direct replacement for a single global lock. Transactions provide the same atomicity and isolation guarantees as the GIL provides for the execution of bytecode instructions. So instead of acquiring and releasing the GIL between these instructions, this approach runs the protected instructions inside transactions.

TM can be implemented in software (STM) or in hardware (HTM). There are also hybrid approaches, which combine the two. We count these hybrid approaches as STM, since they usually provide the same capabilities as software-only approaches but with different performance characteristics. We will now first look at HTM, which recently gained a lot of popularity by its introduction in common desktop CPUs from Intel (Haswell generation) [2, 17].

HTM provides us with transactions like any TM system does. It can be used as a direct replacement for the GIL [2, 6, 7]. However, as is common with hardware-only solutions, there are quite a few limitations that can not be lifted easily. For this comparison, we look at the implementation of Intel in recent Haswell generation CPUs.

[2] There are rare cases where not having atomic bytecodes actually changes the semantics.

[3] https://docs.python.org/2/library/multiprocessing.html

HTM in these CPUs works on the level of caches. This has a few consequences like false-sharing on the cache-line level, and most importantly it limits the amount of memory that can be accessed within a transaction. This transaction-length limitation makes it necessary to have a fallback in place in case this limit is reached. In recent attempts, the usual fallback is the GIL [2, 7]. In our experiments, the current generation of HTM proved to be very fragile and thus needing the fallback very often. Consequently, scalability suffered a lot from this.

The performance of HTM is pretty good as it does not introduce much overhead ($< 40\%$ overhead [2]). And it can transparently parallelise existing applications to some degree. The implementation is very straight-forward because it directly replaces the GIL in a central place. HTM is also directly compatible with any external library that needs to be integrated and synchronised for use in multiple threads. The one thing that is missing is support for a better synchronisation mechanism for the application. It is not possible in general to expose the hardware-transactions to the application in the form of atomic blocks, because that would require much longer transactions.

STM provides all the same benefits as HTM except for its performance. It is not unusual for the overhead introduced by STM to be between 100% to even 1000% [4, 5]. While STM systems often scale very well to a big number of threads and eventually overtake the single-threaded execution, they often provide no benefits at all for low numbers of threads (1-8). There are some attempts [3, 10] that can reduce the overhead a lot, but scale badly or only for certain workloads. Often the benefits on more than one thread are too little in real world applications.

However, STM compared to HTM does not suffer from the same restricting limitations. Transactions can in principle be arbitrarily long. This makes it possible to actually expose transactions to the application in the form of atomic blocks. This is the only approach that enables a better synchronisation mechanism than locks for applications *and* still parallelises when using it. We think this is a very important point because it not only gives dynamic languages the ability to parallelise (already commonplace in most other languages), but also pushes parallel programming forward. Together with sequential consistency it provides a lot of simplification for parallel applications.

While one can argue that STM requires the insertion of read and write barriers in the whole program, this can be done automatically and locally by a program transformation [8]. There are attempts to do the same for fine-grained locking [9] but they require a whole program analysis since locks are inherently non-composable. The effectiveness of these approaches is doubtful in our use case, since we execute bytecode instructions in any order defined by a script only known at runtime. This makes it close to impossible to order locks consistently or to know in advance which locks a transaction will need.

3. The Way Forward

Following the above argumentation for each approach we assembled a general overview in Table 1. The general picture is everything else than clear. It looks like HTM may be a good solution to replace the GIL in the near future. Current implementations are however far too limiting and do not provide good scaling.

Allowing for parallel execution just means that dynamic languages catch up to all other languages that already provide real parallelism. This is why we think that only the STM approach is a viable solution in the long-term. It provides the application with a simple memory model (sequential consistency) and a composable way to synchronise memory accesses using atomic blocks.

Unfortunately, STM has a big performance problem. Particularly, for our use case there is not much static information available since we are executing a program only known at runtime. Additionally, replacing the GIL means running every part of the application in transactions, so there is not much code that can run outside and that can be optimised better. The performance of the TM system is vital.

One way to get more performance is to develop STM systems that make better use of low-level features in existing OS kernels. We are currently working on a STM system that makes use of several such features like virtual memory and memory segmentation. We further tailor the system to the discussed use case which gives us an advantage over other STM systems that are more general. With this approach, initial results suggest that we can keep the overhead of STM below 50%. A hybrid TM system, which also uses HTM to accelerate certain tasks, looks like a very promising direction of research too. We believe that further work to reduce the overhead of STM is very worthwhile.

Acknowledgments

We would like to thank Maciej Fijalkowski and Carl Friedrich Bolz for their valuable inputs and the many fruitful discussions.

References

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

[2] Odaira, Rei, Jose G. Castanos, and Hisanobu Tomari. "Eliminating global interpreter locks in Ruby through hardware transactional memory." *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 2014.

[3] Wamhoff, Jons-Tobias, et al. "FastLane: improving performance of software transactional memory for low thread counts." *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 2013.

[4] Dragojević, Aleksandar, et al. "Why STM can be more than a research toy." *Communications of the ACM* 54.4 (2011): 70-77.

[5] Cascaval, Calin, et al. "Software transactional memory: Why is it only a research toy?." *Queue* 6.5 (2008): 40.

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

[7] Fuad Tabba. 2010. Adding concurrency in python using a commercial processor's hardware transactional memory support. *SIGARCH Comput. Archit. News 38*, 5 (April 2010)

[8] Felber, Pascal, et al. "Transactifying applications using an open compiler framework." *TRANSACT*, August (2007): 4-6.

[9] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: synchronization inference for atomic sections. *In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*. ACM, New York, NY, USA

[10] Spear, Michael F., et al. "Transactional mutex locks." *SIGPLAN Workshop on Transactional Computing.* 2009.

[11] Lamport, Leslie. "How to make a multiprocessor computer that correctly executes multiprocess programs." *Computers, IEEE Transactions* on 100.9 (1979): 690-691.

[12] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A study of transactional memory vs. locks in practice. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (SPAA '11). ACM, New York, NY, USA

[13] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier?. *SIGPLAN* Not. 45, 5 (January 2010), 47-56.

[14] Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. *In Proceedings of the 18th annual ACM SIGPLAN*

conference on Object-oriented programing, systems, languages, and applications (OOPSLA '03).

[15] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. *In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '05).

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

[17] Leis, Viktor, Alfons Kemper, and Thomas Neumann. "Exploiting Hardware Transactional Memory in Main-Memory Databases." *Proc. of ICDE*. 2014.