Ive implemented the hierarchical keys, and it works nicely. This enables a complete storage of keys, eliminating worries about potential hash collisions, without having too much duplicated key information in the database.

Ive (belatedly) come to realize another important thing: pickle serialization is not deterministic. For instance, two identical dicts may have a different pickles depending on their insertion order. That isn’t such a problem for local caching, but it is a fatal flaw for pycc type integration. But this appears to me a solvable problem; my attempt at addressing this can be found in the serialization.py module.

Ive placed the code on github by the way:

<https://github.com/EelcoHoogendoorn/cachepy>

In general, review is much appreciated!

To summarize the current state of affairs: Here is how I envision this system could be used. Note that you could insert caching at any point in the workflow, but it takes some judgment to find the optimal insertion point(s). We want to cache expensive parts of the workflow, but preferably also hook into a point in our flow of data which is natural from the perspective of the cache:

Say we want to cache a function mapping an intermediate code representation and a set of JIT-arguments to a binary code string ready for importing into python, like this:

Class CompliationCache(AbstractCache):

def operation(self, generic\_code\_representation, jit\_arguments):

#bind jit arguments to generic code to get valid C code

#write C code files to temp directory

#call compiler

#load generated binary code from compiler output dir and return

From the perspective of caching the cost of running the compiler, the logical thing to do is to hook up the cache after the first line (which binds the jit arguments, and need not be cached on performance grounds). However, the code representation and its relevant dependencies is a relatively gigantic object, relative to the jit arguments. Implementing it in this way allows the cache to exploit the situation where the same group of code files is compiled many times for different jit arguments, while the code to be compiled has to be stored as a key in the database only once. Note that the ordering of arguments to the cached operation is relevant here; the least variable arguments should come first for partial key reuse to come into effect.

Note that the environment function for this cached operation should encompass both the state of the external compiler, as well as the relevant state responsible for the internal transformations (the template instantiation source and all its dependencies). This would look conceptually like so:

def environment(self):

#load all numba source that has relevance to the binding of jit arguments to intermediate code. for maintainability, it is probably best to use the whole numba directory, filtered against known-to-be-irrelevant modules. cachepy maybe should include some utility functions to that effect, as all applications will need similar mechanisms

#load the llvm compiler version from disk, or whatever information you feel is sufficient to characterize the state of your compiler. if you want to be perfectly paranoid, you could include your entire compiler bin directory and standard library as well. If your compiler does indeed not change, the database stores this information only once anyway.

#return all relevant data

If the environment function is implemented well, the cache should always produce correct results; that is, the same result as if there was no cache. A validation flag can be enabled on the cache to assert this is true, such as to debug both the caching code, as well as any given environment specification.

Future improvements

* Pycc integration. Any process which is cached in this manner is a possible candidate for pycc type caching and distribution, freeing the end user from having to perform these calculations; and more importantly, from the dependencies they entail. Note that this requires a different environment specification. As a developer, we want to be sure that our cache output is actually representative of the current state of our code. As a user, we don’t care about the current state of our code, or if we have any such code installed in the first place. As such, for pycc type purposes the environment should be set to None, or some minimum of information which may both be relevant and knowable on the deployed system, like ‘win64’. That said, I am not familiar with the design of pycc; if someone who is familiar with it could comment further on the integration of pycc with a general caching mechanism, thatd be great. As a sidenote: once we are done building our cache and know all its content, its easy to check for hash collisions, and if none are present, we can simply omit all key data. Given that this is a read-only cache, all locking mechanisms are nothing but noise either; a simple python shelve would suffice. If the static pycc cache and a dynamic local cache should really have the same underlying data store may be questioned. Storing 100mb of keydata locally to eliminate a possibility of hash collisions sounds like a bargain to me; but downloading the same 100mb of what is by now verifiably useless information is probably a harder sell to most. Yet still, there is a strong case to be made for integration. Conceptually, both caches hook into the same codepaths. As such, auto-generating a trimmed pycc cache for distribution from the dynamic cache based on a set of representative code runs could be a very useful feature, for instance.
* NFS compatibility: Ive looked into adding NFS compatibility, and it should be easy to integrate into this design. See for instance <http://stackoverflow.com/questions/668336/platform-independent-file-locking>

A basic lockfile system should be fine for our needs; while this isn’t exactly state of the art database design, for the handful of write transactions per second we are looking at, there shouldn’t be any problem, and it can easily be made to work robustly in a platform independent manner. If I understand the issues correctly, the existing package pylockfile implements all requisite nuances. That said, I don’t have the platform to test any of this; nor the experience to speak with confidence on the matter.

I have at least one unanswered question on this front, for starters: is it safe to read an unrelated key from sqlite3 while another node on an NFS system is writing to a different key? Unless proven otherwise; Id assume not. This implies locking for every single read, which isn’t ideal. Alternatively, we could have a separate read and write lock set up to allow concurrent reads, but not concurrent reads and writes. But this may be premature optimization; also the amount of read transactions isn’t ever going to be big. Though on a large enough cluster… ill leave these optimizations to those with the experience, motivation, and hardware to implement and test them properly; for now ill just note that NFS is very well compatible with the direction I am proposing here.

* Database cleaning? Not sure it adds much. The cache is placed in the os temp directory; in the unlikely event that the cache does get big in a meaningful sense of the word, the os will clean the cache just fine, albeit in not the most intelligent manner. We could add a metadata table though, storing how long it took us to compute the cached values, when we computed them, and how often they have been recalled since. Combined with an entry’s requisite disk storage, this should allow intelligent pruning of the database. Possible to add, but not high priority id say.