Ive implemented the hierarchical keys, and it seems to work out as planned.

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 judgement to find the optimal ones. We want to cache expensive parts of the workflow, but preferably also hook into choke-points in our data flow:

Say we have a function mapping an intermediate code representation and a set of JIT-arguments to a binary object ready for importing into python.

def operation(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 and return generated binary code

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 binding the jit arguments. However, the generic code representation is a relatively large object, and the jit arguments are just a small dict. 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, and 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 key reuse to come into effect.

Note that the environment function for this cache should encompass both the state of the external compiler, as well as the relevant state of the internal passes, such as the template instantiation

def environment(self):

#load all numba source that has relevance to the binding of jit arguments to intermediate code

#load the llvm compiler version from disk, or whatever information you feel is sufficient to characterize the state of your compiler

If the environment function is implemented well, the cache should always produce correct results, in the sense that any change in the environment automatically invalidates cache entries.

Future improvements

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, 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 information, when we computed it, and how often it has been recalled since. Combined with its requisite disk storage, this should allow intelligent pruning of the database.

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 moreover, from the dependencies they entail. Note that this requires a different environment specification. As a developer, we want to be sure that our 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 and intent 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.

NFS compatibility:

Ive looked into adding NFS compatibility, and it should be easy to integrate into this design.

<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.

if I understand the issues correctly, the existing 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 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; I guess 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 Id like to note that this appears to be very well within the realm of possibility.