Adding or extending a family of adaptive instructions.

Families of instructions

The core part of PEP 659 (specializing adaptive interpreter) is the families of instructions that perform the adaptive specialization.

A family of instructions has the following fundamental properties:

- It corresponds to a single instruction in the code generated by the bytecode compiler.
- It has a single adaptive instruction that records an execution count and, at regular intervals, attempts to specialize itself. If not specializing, it executes the non-adaptive instruction.
- It has at least one specialized form of the instruction that is tailored for a particular value or set of values at runtime.
- All members of the family must have the same number of inline cache entries, to ensure correct execution. Individual family members do not need to use all of the entries, but must skip over any unused entries when executing.

The current implementation also requires the following, although these are not fundamental and may change:

- All families uses one or more inline cache entries, the first entry is always the counter.
- All instruction names should start with the name of the non-adaptive instruction.
- The adaptive instruction should end in `_ADAPTIVE`.
- Specialized forms should have names describing their specialization.

Example family

The `LOAD_GLOBAL` instruction (in Python/ceval.c) already has an adaptive family that serves as a relatively simple example.

The `LOAD_GLOBAL_ADAPTIVE` instruction performs adaptive specialization, calling `_Py_Specialize_LoadGlobal()` when the counter reaches zero.

There are two specialized instructions in the family, `LOAD_GLOBAL_MODULE` which is specialized for global variables in the module, and `LOAD_GLOBAL_BUILTIN` which is specialized for builtin variables.

Performance analysis

The benefit of a specialization can be assessed with the following formula: `Tbase/Tadaptive`.

Where `Tbase` is the mean time to execute the base instruction, and `Tadaptive` is the mean time to execute the specialized and adaptive forms.

`Tadaptive = (sum(Ti*Ni) + Tmiss*Nmiss)/(sum(Ni)+Nmiss)`

`Ti` is the time to execute the `i` th instruction in the family and `Ni` is the number of times that instruction is executed. `Tmiss` is the time to process a miss, including de-optimzation and the time to execute the base instruction.

The ideal situation is where misses are rare and the specialized forms are much faster than the base instruction. `LOAD_GLOBAL` is near ideal, `Nmiss/sum(Ni) ≈ 0`. In which case we have `Tadaptive ≈ sum(Ti*Ni)`. Since we can expect the specialized forms `LOAD_GLOBAL_MODULE` and `LOAD_GLOBAL_BUILTIN` to be much faster than the adaptive base instruction, we would expect the specialization of `LOAD_GLOBAL` to be profitable.

Design considerations

While `LOAD_GLOBAL` may be ideal, instructions like `LOAD_ATTR` and `CALL_FUNCTION` are not. For maximum performance we want to keep `Ti` low for all specialized instructions and `Nmiss` as low as possible.

Keeping `Nmiss` low means that there should be specializations for almost all values seen by the base instruction. Keeping `sum(Ti*Ni)` low means keeping `Ti` low which means minimizing branches and dependent memory accesses (pointer chasing). These two objectives may be in conflict, requiring judgement and experimentation to design the family of instructions.

The size of the inline cache should as small as possible, without impairing performance, to reduce the number of `EXTENDED_ARG` jumps, and to reduce pressure on the CPU's data cache.

Gathering data

Before choosing how to specialize an instruction, it is important to gather some data. What are the patterns of usage of the base instruction? Data can best be gathered by instrumenting the interpreter. Since a specialization function and adaptive instruction are going to be required, instrumentation can most easily be added in the specialization function.

Choice of specializations

The performance of the specializing adaptive interpreter relies on the quality of specialization and keeping the overhead of specialization low.

Specialized instructions must be fast. In order to be fast, specialized instructions should be tailored for a particular set of values that allows them to:

1. Verify that incoming value is part of that set with low overhead.
2. Perform the operation quickly.

This requires that the set of values is chosen such that membership can be tested quickly and that membership is sufficient to allow the operation to performed quickly.

For example, `LOAD_GLOBAL_MODULE` is specialized for `globals()` dictionaries that have a keys with the expected version.

This can be tested quickly:

- `globals->keys->dk_version == expected_version`

and the operation can be performed quickly:

- `value = entries[cache->index].me_value;`.

Because it is impossible to measure the performance of an instruction without also measuring unrelated factors, the assessment of the quality of a specialization will require some judgement.

As a general rule, specialized instructions should be much faster than the base instruction.

Implementation of specialized instructions

In general, specialized instructions should be implemented in two parts:

1. A sequence of guards, each of the form `DEOPT_IF(guard-condition-is-false, BASE_NAME)`.

2. The operation, which should ideally have no branches and a minimum number of dependent memory accesses.

In practice, the parts may overlap, as data required for guards can be re-used in the operation.

If there are branches in the operation, then consider further specialization to eliminate the branches.

Maintaining stats

Finally, take care that stats are gather correctly. After the last `DEOPT_IF` has passed, a hit should be recorded with `STAT_INC(BASE_INSTRUCTION, hit)`. After a optimization has been deferred in the `ADAPTIVE` form, that should be recorded with `STAT_INC(BASE_INSTRUCTION, deferred)`.