CS 263 MyPy
A modern C++17 Python Runtime

By Gareth George and John Thomason

Our Project

- CPython leaves room for improvement: direct threading, a GC without reference counting, jit, etc.
- Unpleasant implementation written in C, we leverage C++17 features to generate efficient code for us
- Modern Python runtime written in C++17
- Experiment with Interpreter Optimizations
  - Direct threading
  - 1 form of garbage collection - mark and sweep
  - Object recycling to avoid allocations (FrameStates and Tuples)
  - Improved cache locality by passing numeric types on the stack
- Simplified foreign function interface
- Optimized for numeric computing

Modern C++ Featur

- Heavy use of templates to generate code fo
  - Many operations are exactly the same but
  - Automate handling of errors with type spe
  - Generating type-specific instantiation of te
    and type checks beyond the initial visitor p

```cpp
// Operator overloading
template<typename OT>
void operator()(ValuePyObject& v1, OT& v2) const {
    // Get the attribute for it
    std::tuple<Value,bool> res = value::PyObject::find_attr_in_obj(v
    if(std::get<1>(res)){
        // Call it like a function
        ArgList arglist(v2);
        arglist.bind(v1);
        std::visit(
            call_visitor(frame, arglist),
            std::get<0>(res)
        );
```

```cpp
struct op_neq { // a ≠ b
    template<typename T1, typename T2>
    static bool action(T1 v1, T2 v2) {
        return v1 ≠ v2;
    }

    constexpr const static char* l_attr = "__ne__";
    constexpr const static char* r_attr = "__ne__";
    constexpr const static char* op_name = "≠";
};

struct op_sub { // a - b
    template<typename T1, typename T2>
    static auto action(T1 v1, T2 v2) {
        return v1 - v2;
    }

    constexpr const static char* l_attr = "__sub__";
    constexpr const static char* r_attr = "__rsub__";
    constexpr const static char* op_name = "-";
};
```

Garbage Collection

- Simple mark and sweep pattern
- Specialized heaps for primitive types improves performance
- Objects can be moved in memory if they are not held by a C++ object so future work may include a mark and copy collector
- Subclasses of heap allow objects to be recycled when allocation is expensive
    - FrameStates and Tuples
- Garbage collector is run when memory use doubles since last it was collected

```
16
17      struct Allocator {
18          size_t size_at_last_gc = 32; // 32 bytes or something like that.
19
20          gc_heap<value::List> heap_list;
21          gc_heap<value::Tuple> heap_tuple;
22          gc_heap<const std::string> heap_string;
23          gc_heap<Code> heap_code;
24          gc_heap<value::PyFunc> heap_pyfunc;
25          gc_heap<value::PyObject> heap_pyobject;
26          gc_heap<value::PyClass> heap_pyclass;
27          gc_heap<std::unordered_map<std::string, Value>> heap_namespace;
28      
```

Direc

```
CASE(POP_TOP)
{
    this->check_sta
    this->value_sta
    GOTO_NEXT_OP ;
}
```

```
#define CONTEXT_SWITCH break;
#define CONTEXT_SWITCH_KEEP_PC return;
// Add a label after each case
#define CASE(arg) case op::arg:\
                  arg:
// The hint below means we expect to always be the top frame
// That's not great at all, but this leaves things like BINARY_ADD
// untouched (they onyl create a new frame when they happen to call an overload in a class)
#define CONTEXT_SWITCH_IF_NEEDED \
    if(__builtin_expect(!(this->interpreter_state->cur_frame.get() == this),0)) break;

#define DECODE_AND_JUMP \
instruction = code->instructions[this->r_pc];\
bytecode = instruction.bytecode;\
arg = instruction.arg;\
EMIT_PER_OPCODE_TIME\
DEBUG("%03llu EVALUATE BYTECODE: %s", this->r_pc, op::name[bytecode])\
goto *jmp_table[bytecode];

#define GOTO_NEXT_OP \
this->r_pc++;\
DECODE_AND_JUMP

// Basically the same, but without incrementing pc
// Used in jumps
#define GOTO_TARGET_OP \
DECODE_AND_JUMP
```

```
const static void* jmp_table[] =
    &&NOP,
    &&POP_TOP,
    &&ROT_TWO,
    &&ROT_THREE,
    &&DUP_TOP,
    &&DUP_TOP_TWO,
    &&UNARY_POSITIVE,
    &&UNARY_NEGATIVE,
    &&UNARY_NOT,
    &&UNARY_INVERT,
    &&BINARY_MATRIX_MULTIPLY,
    &&INPLACE_MATRIX_MULTIPLY,
    &&BINARY_POWER,
    &&BINARY_MULTIPLY,
```

bels-As-Values

Recycling Frame States & Tuples

```
void sweep() {
    for (auto itr = this->objects.begin(); itr != this->objects.end();) {
        auto &obj = *itr;
        if (!obj.flags) { // object.flags must be all 0's for us to clear it :)
            auto next_itr = itr;
            next_itr++;
            // we splice the object out rather than delete it
            (*itr).object.initialize_fields();
            this->freelist.splice(this->freelist.begin(), this->objects, itr);
            itr = next_itr;
        } else {
            (*itr).flags &= ~(ptr_t::FLAG_MARKED);
            ++itr;
        }
    }
}
```

Substantial performance boost as Tuples and FrameStates are the most frequently allocated objects

- Iterators frequently use tuples
- FrameStates created for every function call

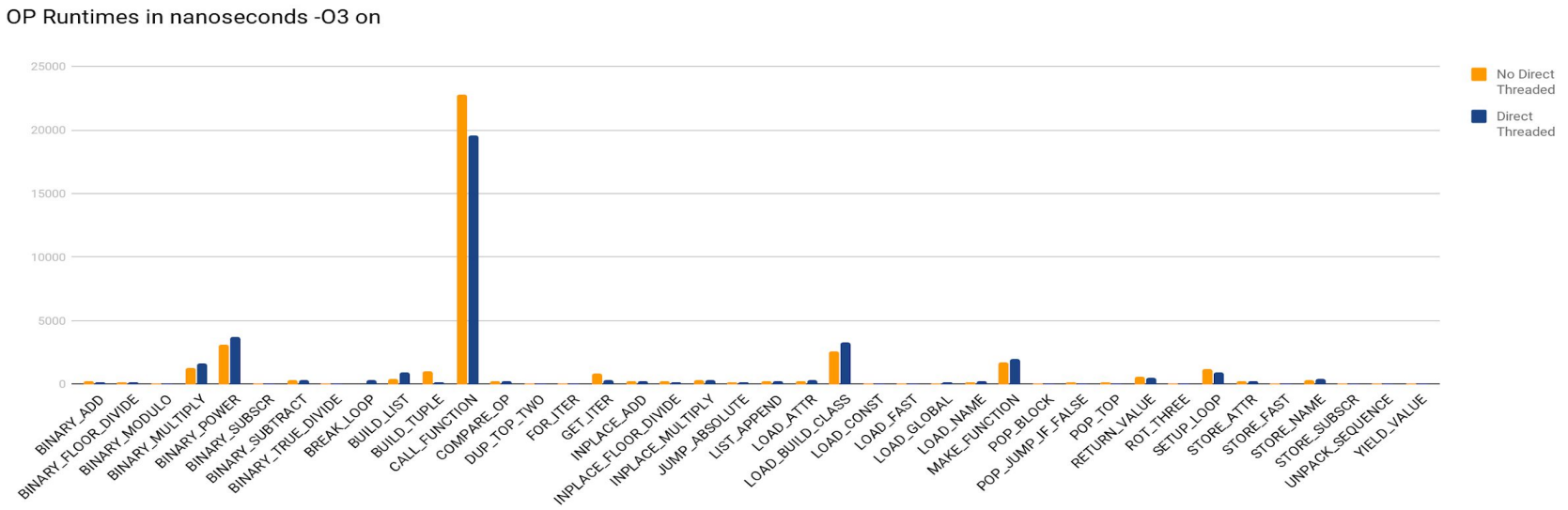
```
ptr_t make(Args&& ... args) {
    using itertype = typename std::list<gc_object>::iterator;
    if (freelist.size() == 0) {
        DEBUG("allocating new memory");
        itertype object_itr =
            this->objects.emplace(this->objects.begin(), std::forward<Args>(args) ... );
        return ptr_t(*object_itr);
    } else {
        DEBUG("recycling allocated memory");
        // always recycle the most recently returned object,
        // its memory is more likely to be in the page cache
        gc_object& obj = this->freelist.front();
        obj.object.recycle(std::forward<Args>(args) ... );
        this->objects.splice(this->objects.begin(), this->freelist, this->freelist.begin());
        return ptr_t(obj);
    }
}
```

When recycling we always reuse the last free’d object first in the hope that it still resides in the CPU’s cache

Drawbacks to Our Implementation

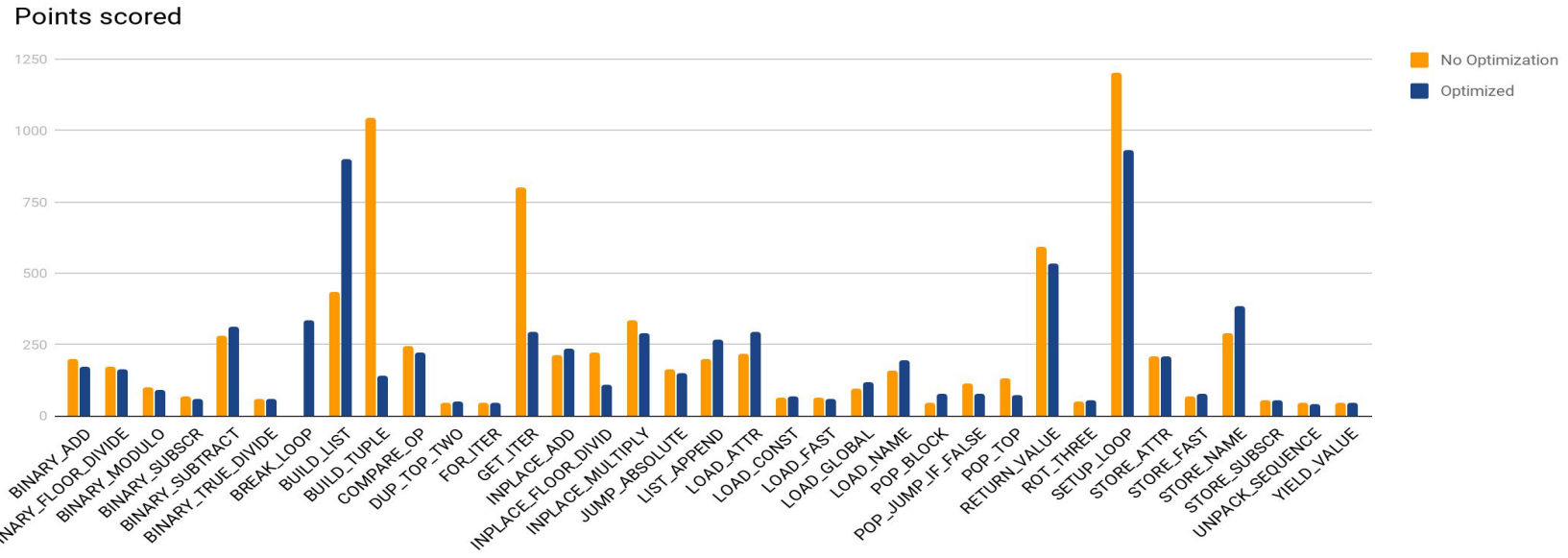
- CALL_FUNCTION slower than we would like
  - Complex logic for setting up arguments
  - Requires allocating and initializing a FrameState, recycling improves this, but still costly
- LOAD_ATTR must search entire class hierarchy every time
  - Cannot easily cache values of parents - python classes offer few guarantees
    - Parents are extremely mutable
  - Future work will include caching and guarding of field lookups
- Variant makes stack allocated types fast BUT at the cost of bloated code generation due to visitor pattern (-O3 executable ~60MB)
- Not all features implemented yet (super(), many other builtins)
- Simple Garbage Collector (mark and sweep)
  - Complexity of garbage collector limited by decision to use stl types such as std::vector, improvements will require a custom implementation of these classes

Profiling Results - Instruction Durations with and without Direct Threading



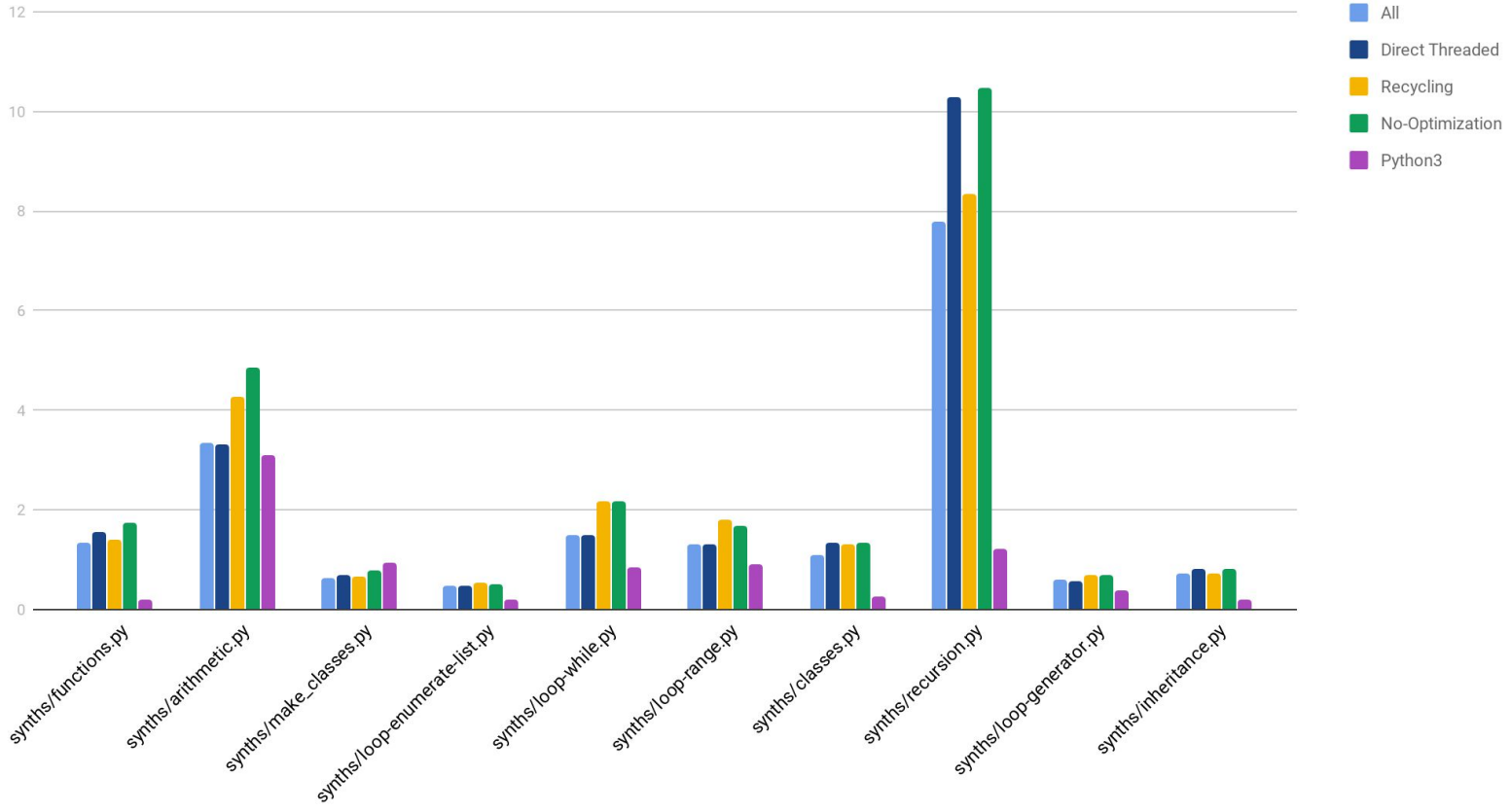
Simple take away -- function calls dominate all other op codes for runtime complexity
They are a major point of future work in performance optimization

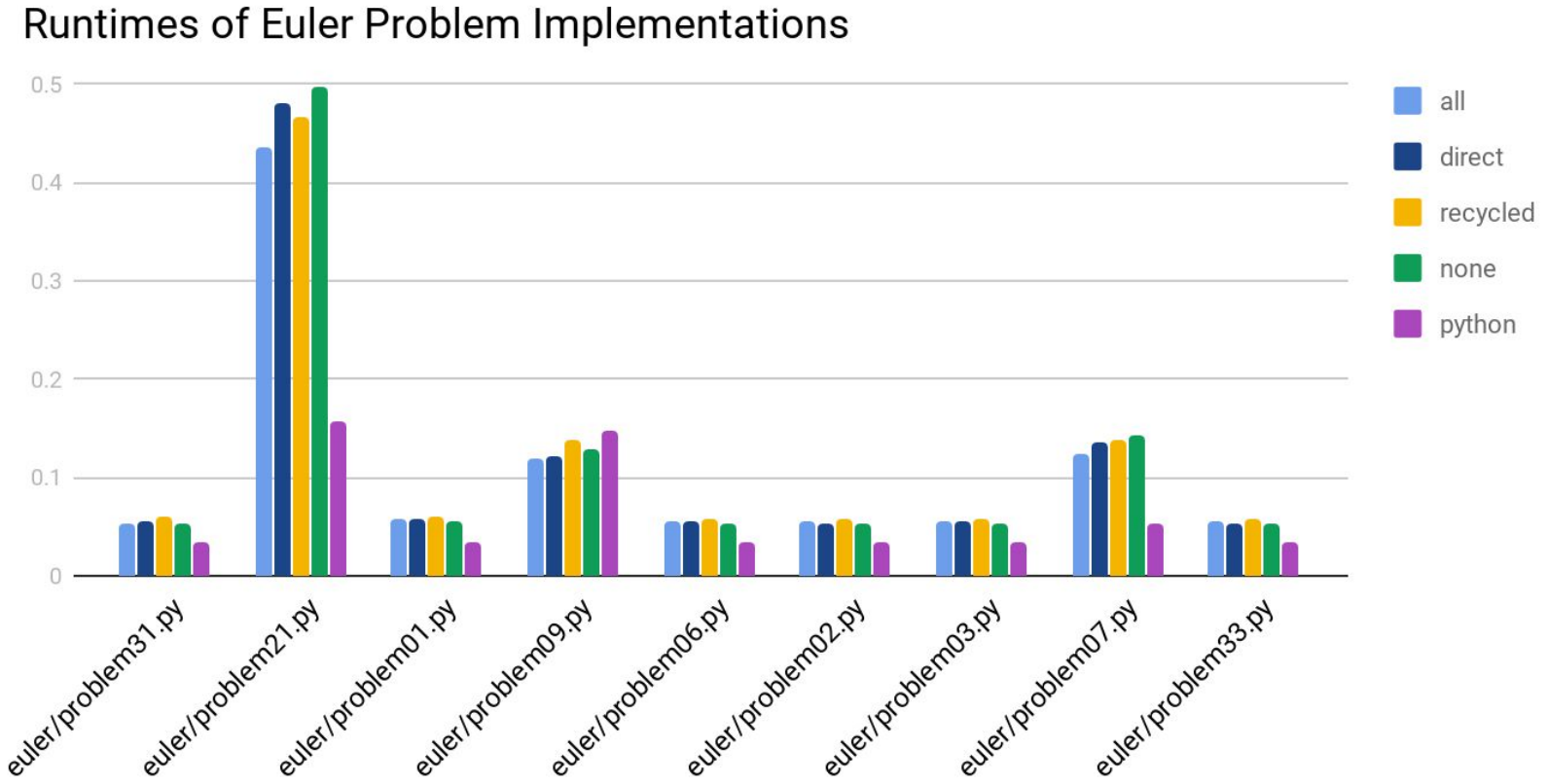
Profiling Results - Instruction Durations with and without Optimization



We can see here that our slowest ops tend to be those that touch large amounts of memory i.e. SETUP_LOOP pushes a block to the infrequently touched BLOCK_STACK (potential to trigger a resize if this is first use).

Runtime Performance in Seconds Across 5 Runs for Various Optimization Settings of MyPy Using Synthetic Benchmarks





We do best in Problem 9 which is purely for range loops and arithmetic, where functions are invoked we do poorly.

Note, We did not implement these solutions, the pure python project euler solutions were taken from https://github.com/jasongros619/Project-Euler

Findings - Performance

- Function calls difficult to make fast,
  - Currently involves passing a vector of arguments - could this be better?
  - Complex argument checking logic involving default arguments, some arguments being involved in closures and some not, and so on - many areas for potential improvement
- Direct threading offers a bump, but not a huge one due to fat op codes (as we would expect)
- Recycling FrameStates, an optimization that CPython also uses, offers a major performance boost

Future Work

- Optimize classes and functions
- Implement JIT for arithmetic expressions
- Finish implementing python features

Questions