Exploring a Tier 2 Interpreter for CPython

Ken Jin† and Jules Poon†

Contributing authors: kenjin@python.org; juliapoopoopoo@gmail.com;

†These authors contributed equally to this work.

Abstract

This report explores removing the overhead associated with dynamic typing in CPython (the reference implementation of Python) using a modified version of lazy basic block versioning [1]. The report also introduces a (to the best of our knowledge) novel type propagation algorithm that is composed of simple type operations on references to types. A speedup of 39% is observed for non-representative workloads purely involving floating-point arithmetic.

1 Introduction

Python is a widely-used programming language [2]. CPython [3] is its reference implementation. Due to Python's dynamic type semantics, CPython is generally unable to execute Python programs as fast as it potentially could with static type semantics.

A technique for removing type check and other overheads associated with dynamic languages is lazy basic block versioning [1]. Lazy basic block versioning is a novel technique for writing *Just-in-Time* (JIT) compilers. This report thus modifies a fork of the CPython 3.12 runtime to implement lazy basic block versioning (without native machine code generation), type propagation, and unboxed float arithmetic, among other optimisations. The modified runtime is called pyLBBV.

Due to the Python language's large scope, pyLBBV only optimises a subset of Python. Specifically, pyLBBV focuses on integer and float arithmetic. We believe this limited subset of Python is enough to showcase pyLBBV's type propagation algorithm and other optimisations.

1.1 CPython Bytecode Architecture Tiers

Execution of CPython is split into logical "tiers". These tiers reflect ascending levels of optimisations and thus execution speed. Tier 0 is generic microcode (termed CPython bytecode) initially emitted by the CPython compiler (Fig. 1).

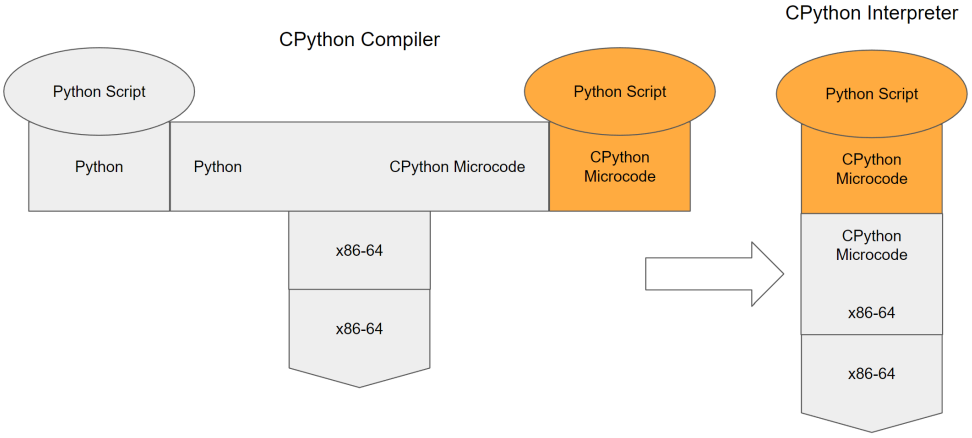

Fig. 1: T-Diagram of Tier 0 CPython Bytecode

Tier 1 is generic bytecode generated by CPython's Specializing Adaptive Interpreter. At runtime, Tier 0 instructions observe common usage patterns and types. Tier 0 instructions then rewrite themselves to Tier 1 instructions if an instruction matching the pattern is observed. Since this optimisation is speculative, Tier 1 instructions contain guards that fall back onto Tier 0 instructions on failure. Tier 1 instructions are able to rewrite themselves back to their generic Tier 0 counterparts if too many guard failures occur (Fig. 2).

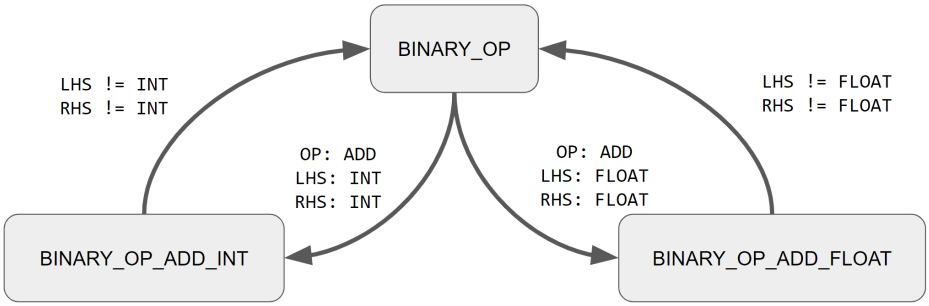

Fig. 2: Instruction state diagram of Tier 0 instruction `BINARY_OP` transitioning to Tier 1 `BINARY_OP_ADD_INT` and `BINARY_OP_ADD_FLOAT` and vice versa.

Tier 1 instructions are only capable of speculative optimisation and exploit single-instruction regions of type stability. They are not capable of doing large-scale optimisations as seen in a statically typed language.

pyLBBV introduces a new tier of instructions - Tier 2. Tier 2 instructions are relatively lower-level than Tier 1 instructions. Each instruction is less complex and executes less code. Tier 1 instructions can be viewed as being composed of multiple Tier 2 instructions. Tier 2 instructions have typed variants that operate only on specific types. Tier 2 instructions and their benefits will be further explained in subsequent sections.

1.2 Goals

Our base goals:

1. Implement the lazy basic block versioning infrastructure.
2. Implement type propagation.
3. Augment CPython's DSL to support type propagation rules.
4. Implement type guard elimination for arithmetic integer and float operations.
5. Run benchmarks from CPython's pyperformance benchmark suite.

We achieved all of our base goals except for running benchmarks. We only ran a single benchmark (the `nbody` benchmark) from CPython's `pyperformance` benchmark suite. We were unable to run more benchmarks due to time constraints.

Our reach goals:

1. Implement type guard elimination for container types (`list`, `tuple`, etc.).
2. Implement interprocedural type propagation.
3. Implement unboxed arithmetic for doubles.
4. Run even more benchmarks from CPython's `pyperformance` benchmark suite.

We achieved some of our reach goals. Namely, we achieved type guard elimination for one container type (`list`), and unboxed arithmetic for doubles. We were not able to do interprocedural type propagation or run additional benchmarks from `pyperformance`.

2 Lazy Basic Block Versioning Tier 2 Architecture Overview

The C structure representing executable CPython bytecode is `PyCodeObject`. In pyLBBV, each `PyCodeObject` has a *hot counter* that starts at -64. Each execution of a backwards jump or the `PyCodeObject`'s entry point increments the hot counter by 1. Hence the hot counter accounts for "hot" loops in a `PyCodeObject`, not just the number of times the `PyCodeObject` is called. When the hot counter reaches 0, pyLBBV attempts to generate Tier 2 bytecode on the next entry to the `PyCodeObject`.

By the time the *hot counter* reaches 0, Tier 1 specialisation would have triggered. So upon the next entry to the `PyCodeObject`, pyLBBV searches for *optimisable* bytecode, which are bytecodes which we have Tier 2 equivalents for and are also Tier 1 specialised (E.g., `int` or `float` specialised `BINARY_OP`). pyLBBV uses Tier 1 specialisation as a proxy for whether the types encountered are primitive (E.g., `int` or `float`), as the absence of Tier 1 specialised bytecode would indicate that the `PyCodeObject` failed to encounter primitive types. If no optimisable bytecode is found, pyLBBV does not generate Tier 2 bytecode, and instead falls back to Tier 1 bytecode.

The *type context*, the data-structure the type propagator uses to keep track of type information, is then initialised. Tier 1 bytecode is then copied into the `PyCodeObject`'s Tier 2 instruction array until pyLBBV hits a branch or an optimisable bytecode. The type propagator propagates the type context through each emitted instruction. If it is a branch, pyLBBV emits a branch instruction which contains information on the next Tier 1 instruction to start the next basic block from. These branch instructions supports the lazy generation of basic blocks during execution of bytecode. See 4.2.1 for more details on jump rewriting. Executing the branch instruction triggers the generation of the target branch and this process repeats until the execution hits a *scope exit*, an instruction that exits the `PyCodeObject` (E.g., `RETURN_VALUE`). One benefit of directly copying Tier 1 instructions is that their inline cache entries are preserved, thus their previous specialisation attempts are not wasted.

If pyLBBV encounters an optimisable bytecode and the types of its operands are unknown, pyLBBV emits a *type guard*, an instruction that checks the types of variables on the stack, followed by a branch instruction that determines if the type guard has passed. Similar to above, the runtime then runs the newly emitted bytecode until it hits the branch instruction.

If the type guard passed, the type propagator then updates the type context with the typing information gained via the type guard, and generates a type-specialised version of the target basic block (e.g, guard elimination, unboxing). See 4.3 for more details. Otherwise, the runtime emits a new type guard and a branch instruction. See 4.1.1 for more details. Here, the type propagator is used to inform the Tier 2 code generator whether it can **guarantee** a variable is of a certain type, and hence which optimisations can be performed.

Furthermore, as will be illustrated below, the type guards result in potentially multiple "versions" of a basic block in Tier 1, each version specialised to an encountered type context. Due to the lazy nature of this generation, branches not taken or type context never encountered will not trigger the generation of a Tier 2 basic block. This allows for automatic dead code elimination.

Eventually the `PyCodeObject` encounters all the type contexts it needs to handle, and will *stabilise* (no new Tier 2 code generated). The code generator and type propagator will now no longer be invoked on this `PyCodeObject` for any subsequent executions (assuming the types observed at runtime do not change).

2.1 Example

```
1  def f(a,b):
2    return a+b+a
3
4  for _ in range(63): f(1.0, 1.0)
5  f(1.0, 1.0)
6  f(1, 1)
```

Tier 2 generation is illustrated in the above code snippet. Note that at line 5 (Fig. 3), Tier 2 is triggered on `f` with the types `{a: float, b: float}`. At line 6, `f` is called with types `{a: int, b: int}` (Fig. 4).

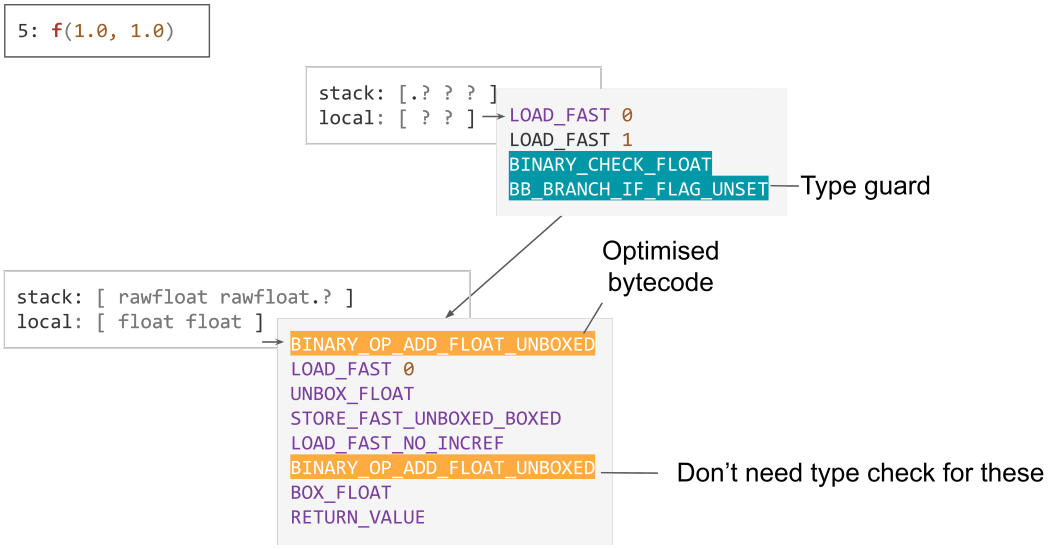

Fig. 3: Tier 2 bytecode generation during the execution of line 5

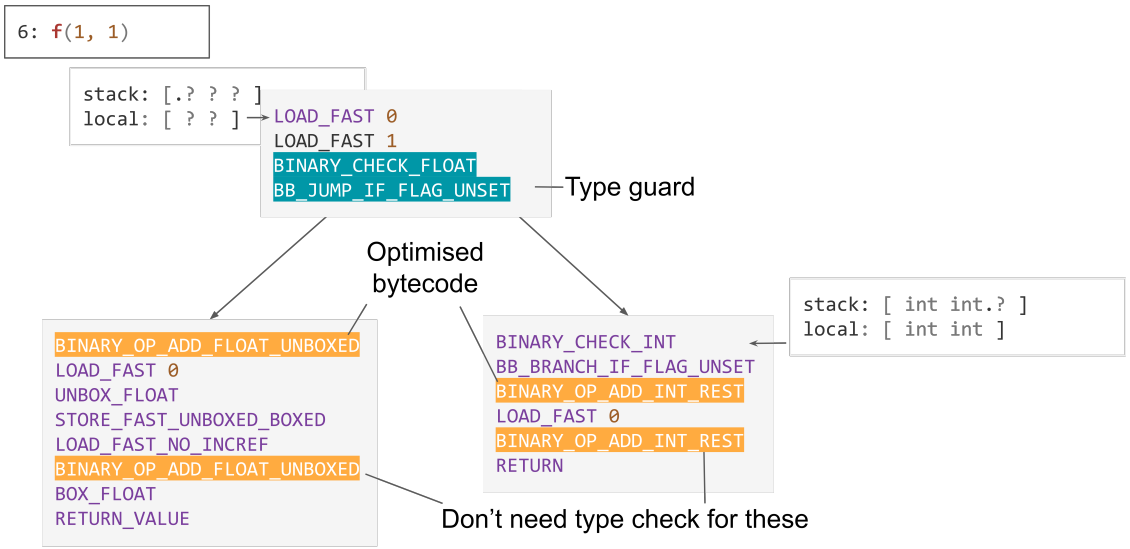

Fig. 4: Tier 2 bytecode generation during the execution of line 6

At line 5, after the two `LOAD_FAST` are emitted, a `BINARY_OP` instruction is encountered and the code generator emits a `BINARY_CHECK_FLOAT`. The runtime then executes until the type guard,

which passes. The type propagator then updates the type context and emits the rest of the bytecode, unboxing the float (see 4.3.2) and eliminating the type-guard in the subsequent `BINARY_OP`. The runtime then executes the newly emitted bytecode until scope exit (E.g. `RETURN_VALUE`).

At line 6, the type guard `BINARY_CHECK_FLOAT` fails, triggering `BB_BRANCH_IF_FLAG_UNSET` to rewrite itself to `BB_JUMP_IF_FLAG_UNSET` and the code generator to emit another type guard `BINARY_CHECK_INT`. The runtime then executes the new type guard which passes, and the type propagator updates the type context and similarly emits the rest of the bytecode.

Any subsequent evaluation of `f` on `(float,float)` or `(int,int)` will not result in any more new Tier 2 bytecode.

3 Type Propagator

To recap, the type propagator is used during the generation of Tier 2 bytecode to inform the code generator whether we can guarantee a variable is of a certain type. The code generator then uses this information to emit the next bytecode(s) and calls the type propagator to propagate over the newly emitted bytecode.

This report defines the data structure from which we store the type information of the CPython `PyCodeObject` as the *type context*. The type context mirrors the CPython run-time state during execution (e.g., the operand stack pointer `stack_ptr` at the current instruction matches the `stack_ptr` of the type context when propagating across said instruction). The type propagator hence "interprets" the bytecode. Each Basic Block stores a type context at its exit to support the lazy generation of bytecode and backwards-jump targets store a type context at their entry. See Section 4 for more details.

3.1 Type Context

Mirroring CPython's `PyCodeObject`, the type context contains a stack (`type_stack`) and a local variables array (`type_locals`). The type propagator does not include an analogue for the constant array (`co_const`) and instead directly queries the executing `PyCodeObject`'s `co_const` for type information of constants, which is possible as `co_const` is never written to.

Each item in the `type_stack` and `type_locals` is a *node* of a *tree*. Each node has a unique parent but can have multiple children, each of which is another node in the same type context. Trees are a useful construct in relating two nodes that are guaranteed to be of the same type: If two nodes are in the same tree, they are the same type. The tree's *root* is used to determine the type of the entire tree.

```
// Retrieved from /Include/cpython/code.h
typedef struct _PyTier2TypeContext {
    // points into type_stack, points to one element after the stack
    _Py_TYPENODE_t *type_stack_ptr;
    int type_locals_len;
    int type_stack_len;
    _Py_TYPENODE_t *type_stack;
    _Py_TYPENODE_t *type_locals;
} _PyTier2TypeContext;
```

The type context data structure `_PyTier2TypeContext` (above) is very simple memory-wise, consisting mainly of two arrays `type_stack` and `type_locals`. Each node (`_Py_TYPENODE_t`) is a tagged pointer. Since pyLBBV can assume pointer alignment, the lower 2 bits of each pointer are used to tag a `_Py_TYPENODE_t`:

```
// Retrieved from /Include/cpython/code.h
typedef enum _Py_TypeNodeTags {
    // An invalid node without a tag, kept for catching bugs
    TYPE_NULL = 0,
    // The root of a tree.
```

```
    // TYPE_ROOT can point to a PyTypeObject or be a NULL
    TYPE_ROOT = 1,
    // A non-root of a tree (has a parent).
    // TYPE_REF points to a TYPE_ROOT or a TYPE_REF
    TYPE_REF  = 2
} _Py_TypeNodeTags;
```

The entire tree can be encoded within tagged pointers. Managing the memory of these trees is hence very simple, requiring only a single allocation and de-allocation through the lifetime of a type context.

Accessing the parent of a node only involves a dereference. However, enumerating the children of a node requires traversing the entire `type_stack` and `type_locals` arrays. In practice, these arrays are small and a linear search suffices.

3.1.1 Design Motivation

The reason for representing variables as nodes of trees is to encode relationships between variables. Take for instance:

```
def f(a,b):
  return a+b
f(1,1)
```

which generates the following trace:

```
offset: 1
  [-] Type propagating across: LOAD_FAST : 0
      Stack: 000001F5B5C096C0: [ ?->locals[0]]
      Locals 000001F5B5C096F0: [ ? ?]
offset: 2
  [-] Type propagating across: LOAD_FAST : 1
      Stack: 000001F5B5C096C0: [ ?->locals[0] ?->locals[1]]
      Locals 000001F5B5C096F0: [ ? ?]
offset: 3
  [-] Type propagating across: BINARY_CHECK_INT : 1
      Stack: 000001F5B5C09780: [ int->locals[0] int->locals[1]]
      Locals 000001F5B5C09720: [ int int]
offset: 3
  [-] Type propagating across: BINARY_OP_ADD_INT_REST : 0
      Stack: 000001F5B5C09780: [ int]
      Locals 000001F5B5C09720: [ int int]
```

The two `LOAD_FAST` loads both locals onto the stack and `BINARY_CHECK_INT` checks if the last two stack variables are integers. If they are, then the type propagator also knows that the two local variables are integers as well.

This "propagation" of type information is crucial for CPython bytecode, as suppose otherwise that `BINARY_CHECK_INT` does not result in the propagator knowing that the local variables are integers. Then the subsequent `BINARY_OP_ADD_INT_REST` will overwrite the first stack variable with another integer representing the result, and subsequent instructions could potentially overwrite the second stack variable as well. Any subsequent `LOAD_FAST 0` and `LOAD_FAST 1` will require a type check again, rendering the previous `BINARY_CHECK_INT` type check useless.

3.2 Type Operations

The type propagator implements propagation rules for each CPython bytecode instruction, where each instruction modifies the trees in the type context. In pyLBBV, the propagation rules for most instructions can be accurately modelled with 3 type operations in order of frequency: `TYPE_OVERWRITE`, `TYPE_SET` and `TYPE_SWAP`. Each operation has the rough syntax of `type_operation(src, dst)`, where the source (`src`) and destination (`dst`) are nodes. Note that `src` node might not be in the type context as in the case of "introducing"

a new root. E.g., `GET_LEN` pushes an integer (`PyLong_Type`) onto the stack, whereby the `src = set_root_tag(&PyLong_Type)` is a root node not part of the type context.

3.2.1 `TYPE_OVERWRITE`

`TYPE_OVERWRITE(src, dst)` is an operation to model an "overwriting". `dst` gets overwritten by `src` so `dst` now has the same type as `src` and the old `dst` node no longer exists. In terms of trees, the type propagator is removing `dst` for its tree and making `dst` a child of `src`. These present 4 cases:

1. If `dst` is a root and `src` is in the type context, set one of the children of `dst` as the new root and set `dst` to be a child of `src`.
2. If `dst` is not a root and `src` is in the type context, set all of the children of `dst` to point to `dst`'s parent and set `dst` to be a child of `src`.
3. If `dst` is a root and `src` is not in the type context, set one of the children of `dst` as the new root and write `src` to `dst`.
4. If `dst` is not a root and `src` is not in the type context, set all of the children of `dst` to point to `dst`'s parent and write `src` to `dst`.

An example of where such an operation is used is in `BINARY_OP_ADD_INT_REST`, where the result of the operation overwrites a stack variable.

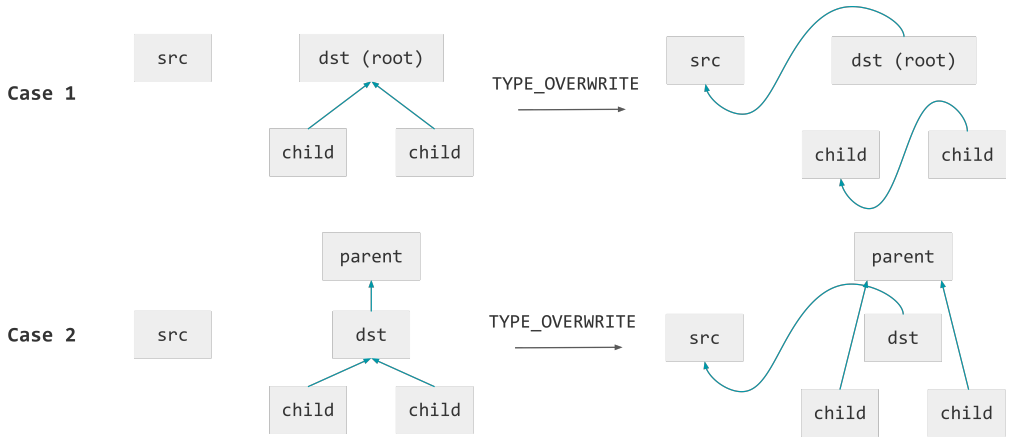

Fig. 5: Case 1 and 2 of `TYPE_OVERWRITE` illustrated

3.2.2 `TYPE_SET`

`TYPE_SET(src, dst)` is an operation to model the gain in knowledge that two nodes share the same type. All nodes of `dst`'s tree will have the same type as the `src`'s node. In terms of trees, we take the root of the `dst`'s tree and make it part of the `src` tree, effectively making the `dst`'s tree a subtree of the `src` tree. These similarly presents 2 cases:

1. If `src` is in the type context, set the root of the `dst`'s tree as a child of the `src` node.
2. If `src` is not in the type context, write `src` to `dst`'s tree's root.

An example of where such an operation is used is in `BINARY_CHECK_INT`, where the type propagator gains the knowledge that the top two stack variables are integers.

3.2.3 `TYPE_SWAP`

`TYPE_SET(src, dst)` is an operation to model the swapping of the positions of two nodes within the type context. In terms of trees, if `src` and `dst` are part of the same tree, the type propagator does nothing. Otherwise, the nodes are swapped and the parents and children of both nodes are modified to preserve both trees. The only CPython instruction that utilises this is `SWAP`, which swaps the position of two stack elements.

3.3 Modified Domain Specific Language

The type propagator implements propagation rules for all instructions, and as pyLBBV contains 194 instructions, it is infeasible to write the propagation rules manually.

During the development of CPython 3.12 (the current version in development as of writing), CPython introduced a *Domain Specific Language* (DSL) to generate the interpreter. The DSL serves as a specification for the behaviour of each instruction. This is similar to describing the structural operational semantics of Python.

Since most instruction's type propagation rules can be determined statically, pyLBBV's approach is to augment the DSL with additional semantics to describe the type propagation rules and use the DSL to generate the type propagator. Each instruction's propagation rules can be expressed as a sequence of type operations (`TYPE_OVERWRITE`, `TYPE_SET`, `TYPE_SWAP`), and pyLBBV's modified DSL allows specifying said sequence for each instruction.

These modifications allow for the type propagator to be generated directly from the same DSL definition as CPython bytecode, thus reducing duplication in code. pyLBBV's type propagator is just under 1000 lines of handwritten C code, with 400 lines of C code for extraneous supporting functions. The DSL-generated type propagation for CPython instructions is over 1000 lines of generated C code. This thus presents a significant reduction of developer maintenance burden.

3.3.1 A Brief Introduction to CPython's Interpreter DSL

The DSL syntax for an instruction consists of a *header* and *body* [4]. The header specifies the stack effect and inline cache, while the body specifies the behaviour of the instruction. Take for instance the Tier 1 instruction `BINARY_OP_MULTIPLY_INT`:

```
// Retrieved from /Python/bytecodes.c

// Header start
macro_inst(BINARY_OP_MULTIPLY_INT, (unused/1, left, right -- prod))
// Header end
{
    // Body start
    assert(cframe.use_tracing == 0);
    DEOPT_IF(!PyLong_CheckExact(left), BINARY_OP);
    DEOPT_IF(!PyLong_CheckExact(right), BINARY_OP);
    U_INST(BINARY_OP_MULTIPLY_INT_REST);
    // Body end
}
```

For the header, `unused/1` specifies the size of the inline cache, which follows the format `IDENTIFIER "/" NUMBER`. This affects how much to increment the instruction pointer after executing the instruction. `left` and `right` specifies the input operand stack variables, and `prod` specifies the output operand stack variable(s). The input and output are separated by a `"--"`. Their relative position also tells the interpreter generator at what offsets they are in the operand stack.

Stack variables can also be named `unused`. These variables are by convention, untouched by the body. There are also *size-stack variables*, which follow the format `IDENTIFIER "[" EXPRESSION "]"`, where `EXPRESSION`'s only free variable is `oparg`, the argument to the instruction. Size-stack variables specify a sequence of stack variables. An example of both of these is in the instruction `COPY`, which copies the `(oparg-1)`th element from the top to the top of the operand stack.

```
// Retrieved from /Python/bytecodes.c
inst(COPY, (bottom, unused[oparg-1] -- bottom, unused[oparg-1], top)) {
    assert(oparg > 0);
    top = Py_NewRef(bottom);
}
```

Since the type propagator will only use the header, an introduction to the body is omitted.

3.3.2 Additional DSL Syntax for Specifying Propagation Rules

An instruction's propagation rules consist of two components

1. *localeffect*: How the instruction affects the types of the local variables array
2. *stackeffect*: How the instruction affects the types on the stack

pyLBBV specifies the localeffect at the end of the instruction header in the DSL, and the stackeffect as a collection of *type annotations* at the end of each output stack variable. For instance, here are examples of localeffect and type annotations respectively:

```
inst(STORE_FAST, (value --), locals[oparg] = *value) {...}
inst(BINARY_CHECK_INT, (left, right -- left:<<= PyLong_Type, right:<<= PyLong_Type)) {...}
```

In both the localeffect and type annotations, pyLBBV specifies the `src` and the `dst` in a type operation, `src` being green, `dst` being pink and the operation in cyan. The type operation for localeffect is always a `TYPE_OVERWRITE`.

```
inst(STORE_FAST, (value --), locals[oparg] = *value) {...}
inst(BINARY_CHECK_INT, (left, right -- left:<<= PyLong_Type, right:<<= PyLong_Type)) {...}
```

The `src` for a localeffect can be either a *typeliteral* (e.g., `PyLong_Type`) or an input stack variable, while `src` for a type annotation can either be a typeliteral, input stack variable, an element from the `PyCodeObject`'s locals array or constant array. In this sense, knowledge of types only comes from a type operation with either an element from the constant array or a typeliteral as the source.

Both the localeffect and type annotations for each output stack variable are optional. See Appendix A for how pyLBBV handles those. Localeffect and type annotations have the following EBNF:

```
local_effect := "locals" "[" expr "]" "=" local_source
local_source := localsrc_literal | localsrc_stackinput
localsrc_literal := identifier
localsrc_stackinput := "*" identifier

type_annotation := ("{" type_operation ( "," type_operation )... "}" )
    | type_operation
type_operation := type_set | type_overwrite
type_set := "<<=" type_source
type_overwrite := type_source

type_source := typesrc_literal
    | typesrc_locals
    | typesrc_const
    | typesrc_stackinput
typesrc_literal := identifier
typesrc_locals := "locals" "[" expr "]"
typesrc_consts := "consts" "[" expr "]"
typesrc_stackinput := "*" identifier
```

From the EBNF, pyLBBV can formally specify how the added semantics will generate the type propagator by specifying the operational semantics which describes the propagator's behaviour. See Appendix A for the full specification.

3.4 Handling Run-time Dependent Stack Effect

The DSL approach works well for instructions whose type propagation rules can be determined statically. Unfortunately, there is a class of branch instructions with runtime-dependent stack effect. E.g., `BB_TEST_ITER` tests if the iterator (`iter`) on the stack has a `next` element, and if so, it pushes

`next` onto the stack. Otherwise, the iterator is exhausted and `BB_TEST_ITER` pops `iter` off the stack.

This presents an issue for pyLBBV as it type propagates before execution. For these class of instructions, since they appear at the end of a basic block and the conditional stack effect also determines the branch that will be taken, pyLBBV type propagates when the target branch's generation is triggered. At that point, pyLBBV knows which branch was taken and hence the stack effect of the instruction. Thus it can modify the type context accordingly before generating the branch.

4 Basic Block Generation

Some Tier 2 bytecode split their Tier 1 counterparts, separating the type checking from the main operation. To prevent code duplication, pyLBBV introduced `macro_inst` and `u_inst` into the DSL, where a `macro_inst` can be inlined with `u_inst`(s). For instance, pyLBBV can split `BINARY_OP_MULTIPLY_INT` as such:

```
// Retrieved from /Python/bytecodes.c
macro_inst(BINARY_OP_MULTIPLY_INT, (unused/1, left, right -- prod)) {
    assert(cframe.use_tracing == 0);
    DEOPT_IF(!PyLong_CheckExact(left), BINARY_OP);
    DEOPT_IF(!PyLong_CheckExact(right), BINARY_OP);
    U_INST(BINARY_OP_MULTIPLY_INT_REST);
}

u_inst(BINARY_OP_MULTIPLY_INT_REST, (left, right -- prod : PyLong_Type)) {
    STAT_INC(BINARY_OP, hit);
    prod = _PyLong_Multiply((PyLongObject *)left, (PyLongObject *)right);
    _Py_DECREF_SPECIALIZED(right, (destructor)PyObject_Free);
    _Py_DECREF_SPECIALIZED(left, (destructor)PyObject_Free);
    ERROR_IF(prod == NULL, error);
}
```

4.1 Lazy Generation

4.1.1 The "Ladder" of Types

In basic block versioning, operations and their type checks are defined in the same language that is being executed (JavaScript). Similar to the lazy basic block Versioning [1] paper, a type check failure generates another type check in the modified runtime. However the key difference in pyLBBV is that pyLBBV does not define type checks in Python itself. Instead, type checks follow a simple ladder of type checks. For example, after a `BINARY_CHECK_FLOAT` failure occurs, a `BINARY_CHECK_INT` is emitted.

Eventually, the end of the ladder of types is reached. When pyLBBV bottoms out, it falls back on Tier 0/1 by emitting a Tier 0/1 instruction. This thus allows pyLBBV to handle code polymorphism and type instability.

```
// Retrieved from /Python/tier2.c
static int type_guard_ladder[256] = {
    -1,
    BINARY_CHECK_FLOAT,
    BINARY_CHECK_INT,
    -1,
    CHECK_LIST,
    -1,
};
```

4.2 Transformations

4.2.1 Jump Rewriting

Since basic blocks of varying lengths are generated on the fly, jump offsets are not known initially. Jump instructions need to be re-written as new basic blocks are generated. pyLBBV thus contains transient self-rewriting Tier 2 jump instructions that modify themselves depending on which successor basic block is generated. The key benefit of this is that this does not require any code passes as the instruction itself is responsible for its own modifications.

CPython has (as of 30/03/2023), 11 different jump instructions. pyLBBV has self-rewriting counterparts for most of these (Table 1).

Table 1: CPython jump instruction transformations

Tier 0/1 Jump Instruction	Tier 2 equivalent
`FOR_ITER`	`BB_TEST_ITER`[1]
`POP_JUMP_IF_FALSE`	`BB_TEST_POP_IF_FALSE`
`POP_JUMP_IF_TRUE`	`BB_TEST_POP_IF_TRUE`
`POP_JUMP_IF_NOT_NONE`	`BB_TEST_POP_IF_NOT_NONE`
`POP_JUMP_IF_NONE`	`BB_TEST_POP_IF_NONE`
`JUMP_BACKWARD`	`BB_JUMP_BACKWARD_LAZY`

All of these instructions set a runtime flag depending on the test, and precede a `BB_BRANCH` instruction. The `BB_BRANCH` instruction is responsible for checking the flag and generating (then jumping to) the successor basic block.

[1]This instruction specialises to other instructions. (See Table 2)

After both outgoing edges of a basic block are generated, the jump offsets are known. Thus the branch instruction rewrites itself back to a simple jump instruction to reduce overhead.

4.2.2 Backward Jump Rewriting

A special class of jumps are backwards jumps. These occur at the end of loop constructs such as `for` and `while` statements. While the offsets of Tier 2 backwards jumps are known (as any basic block prior to the jump should have already been generated), whether it is safe to jump to that jump target is unknown. Previous loop iterations may have modified the type context, thus rendering the entire loop body type-unsafe to jump to.

pyLBBV's solution is similar to that implemented in the original lazy basic block versioning [1] paper. pyLBBV stores metadata about backwards jump targets in the code object.

```
// Retrieved from /Include/cpython/code.h
// Tier 2 info stored in the code object. Lazily allocated.
typedef struct _PyTier2Info {
    // ...
    _PyTier2BBStartTypeContextTriplet **backward_jump_target_bb_pairs;
    // ...
} _PyTier2Info;

// ...

typedef struct _PyTier2BBStartTypeContextTriplet {
    int id;
    _Py_CODEUNIT *tier1_start;
    _PyTier2TypeContext *start_type_context;
} _PyTier2BBStartTypeContextTriplet;
```

pyLBBV contains a custom diff function to calculate the "distance" of two type contexts (in this case, the jump target type context and the current type context at the backwards jump). The diff function allows for type widening, but not narrowing, nor does it allow for interchanging of boxed and unboxed types.

If the "distance" is too large, a new loop entry basic block is generated, using the type context at the backward jump. The backward jump then self-rewrites to a forward jump to jump into the new loop body.

4.2.3 Specialised Tier 2 Iteration Instructions

The Tier 2 `BB_TEST_ITER` iteration instruction further specialises itself depending on the iterator it observes at runtime. This uses the same infrastructure as the Tier 1 Specializing Adaptive Interpreter. See Table 2 for the full list of specialised Tier 2 instructions. In general, these instructions contain code optimised for the specific iterator that is being iterated.

Table 2: `BB_TEST_ITER` transformations

Type observed	`BB_TEST_ITER` rewritten to
`PyRangeIter_Type`	`BB_TEST_ITER_RANGE`
`PyListIter_Type`	`BB_TEST_ITER_LIST`
`PyTupleIter_Type`	`BB_TEST_ITER_TUPLE`

These instructions contain guards, as with their Tier 1 counterparts.

4.3 Optimisations

4.3.1 Guard Elimination

All Tier 1 instructions are speculative. They exploit single instruction regions of type stability. During the course of their lifetime, their operand types may change. Thus, Tier 1 instructions have guards to fall back to generic Tier 0 instructions.

These guards are not free. They require test and branches. Guards may incur some overhead in terms of CPU branch mispredictions. For example, the Tier 1 instruction `BINARY_OP_ADD_INT` looks like this:

```
// Retrieved from /Python/bytecodes.c
macro_inst(BINARY_OP_ADD_INT, (unused/1, left, right -- sum)) {
    assert(cframe.use_tracing == 0);
    DEOPT_IF(!PyLong_CheckExact(left), BINARY_OP);
    DEOPT_IF(Py_TYPE(right) != Py_TYPE(left), BINARY_OP);
    U_INST(BINARY_OP_ADD_INT_REST);
}
```

Note the `DEOPT_IF` C macros. These check that the operands are indeed the type `PyLongObject` (CPython's arbitrary precision integers).

Due to information gathered by type propagation, the runtime can infer when the operands `left` and `right` are guaranteed to be `PyLongObject`. Thus, the runtime can eliminate the guards in the instruction. This is represented by emitting the instruction `BINARY_OP_ADD_INT_REST`, which is logically equivalent to `BINARY_OP_ADD_INT`, excluding the guards.

```
// Retrieved from /Python/bytecodes.c
u_inst(BINARY_OP_ADD_INT_REST, (left, right -- sum : PyLong_Type)) {
    STAT_INC(BINARY_OP, hit);
    sum = _PyLong_Add((PyLongObject *)left, (PyLongObject *)right);
    _Py_DECREF_SPECIALIZED(right, (destructor)PyObject_Free);
    _Py_DECREF_SPECIALIZED(left, (destructor)PyObject_Free);
```

```
    ERROR_IF(sum == NULL, error);
}
```

`BINARY_OP_ADD_INT_REST` thus does not incur the same guard overhead as the original `BINARY_OP_ADD_INT`. The rules used when deciding when to emit a guardless instruction are in Table 3.

Table 3: Guardless Instruction Rules

Type 1	Type 2	Operation(s)	Instruction(s) emitted
Arithmetic Operations			
`PyLong_Type`	`PyLong_Type`	`+-*`	`BINARY_OP_X_INT` [1]

`PyFloat_Type`	`PyFloat_Type`	`+-*`	`UNBOX_FLOAT 0`
			`UNBOX_FLOAT 1`
			`BINARY_OP_X_FLOAT_UNBOXED` [2]

`PyRawFloat_Type`	`PyRawFloat_Type`	`+-*`	`BINARY_OP_X_FLOAT_UNBOXED`
Container Subscript Operations			
`PyList_Type`	`PyLong_Type` (Must be 64-bit long)	`[]`	`X_SUBSCR_LIST_INT` [3]

These are expressed in our code with the `infer_BINARY_OP` and `infer_BINARY_SUBSCR` functions in `Python/tier2.c`.

[1]Where `X` refers to the arithmetic operation being applied (`ADD, SUBTRACT, MULTIPLY`).

[2]These instructions are explained in more detail in 4.3.2.

[3]Where `X` refers to a whether the binary subscript is a store operation (`BINARY, STORE`). This instruction contains a single guard to check that the subscripting index does not exceed the container's length.

One aspect of emitting instructions which are not speculative and guarantee their input types, is that their output types are also guaranteed. This is similar to the simple type inference rules introduced in the CS4215 language Source Typed. They follow the following rule:

$$\frac{\Gamma \vdash O_1 : t_1 \quad \Gamma \vdash O_2 : t_2}{\Gamma \vdash p_1[O_1, O_2] : t_3}[\textbf{Tier2BinaryOp}]$$

Where Γ is the type context, O_1 and O_2 are the operand stack operands, and p_1 is one of the Tier 2 binary operations. t_1 and t_2 are the respective types of the operands, and t_3 is the output type. The type inference rules for some operations are found in Table 4.

Table 4: Basic Type Inference Rules

t_1	t_2	p_1	t_3 emitted
Arithmetic Operations			
`PyLong_Type`	`PyLong_Type`	`+, -, *`	`PyLong_Type`
`PyFloat_Type`	`PyFloat_Type`	`+, -, *`	`PyFloat_Type`
`PyRawFloat_Type`	`PyRawFloat_Type`	`+, -, *`	`PyRawFloat_Type`
List Subscript Operations			
`PyList_Type`	`PyLong_Type` (Must be 64-bit long)	`[]`	`Unknown`

These are expressed in our code with the Tier 2 instruction DSL.

These allow the output types of certain operations to feed information back to the type propagator, and allows for type information to continue accumulating as more Tier 2 instructions are executed, thus eliminating more guards over time.

4.3.2 Loop Peeling

A positive side-effect of how backwards jump are handled (see 4.2.2) is that loop peeling occurs naturally. Multiple loop bodies are generated until the type context of the loop body stabilises.

Backward jumps of the loop bodies of peeled iterations are rewritten to forward jumps to jump into the loop body specialised for the next iteration's type context.

For example, the code in Fig. 6 generates 3 loop bodies (1 more than required due to a slight inefficiency in pyLBBV's implementation).

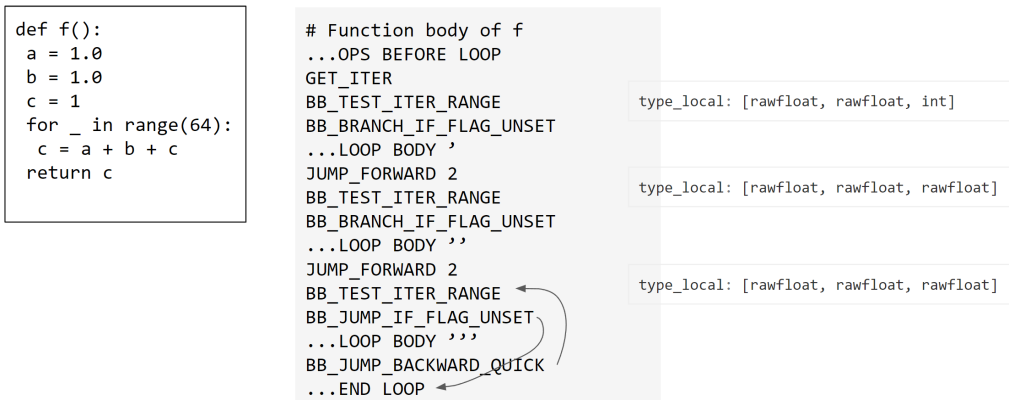

Fig. 6: Generated bytecode for a function with initially unstable types.

This allows for pyLBBV to exploit type stability in loops, and account for type instability in initial iterations of loops.

4.3.3 Double Unboxing

CPython double/float objects are represented as a `PyFloatObject`, which is a simple C structure with a field containing the `double` value itself.

```
// Retrieved from /Include/cpython/floatobject.h
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

Note that CPython does not support auto-promotion of floating-point values (i.e. there's no promotion of `float32` to `float64` to `float128`). Note that CPython also does not support auto boxing and unboxing of `double`s unlike other languages (and their runtimes) like Java. All CPython floating-point arithmetic use the 64-bit C `double`.

With the type propagator and the type inference rules, there is enough information at runtime to guarantee that certain `double` operations are always type safe. The runtime can unbox the `PyFloatObject`s into raw `double` values by simply extracting the value from the `PyFloatObject` and replacing the `PyFloatObject` with a `double` on the operand stack. pyLBBV thus introduces an `UNBOX_FLOAT` instruction. This takes inspiration from the Multi Level Quickening paper which takes a similar approach [5].

The following is the CPython instruction DSL definition of `UNBOX_FLOAT`. Note the type stack effect: it writes a `PyRawFloat_Type` to the type stack at the `oparg`'s offset.

```
// Retrieved from /Python/bytecodes.c
inst(UNBOX_FLOAT, (
    boxed_float, unused[oparg]
    --
    unboxed_float : PyRawFloat_Type, unused[oparg])) {
    double temp = ((PyFloatObject *)boxed_float)->ob_fval;
    Py_DECREF(boxed_float);
    unboxed_float = (*(PyObject **)(&temp));
}
```

This is the code generated and executed at runtime:

```
// Retrieved from /Python/generated_cases.c.h
TARGET(UNBOX_FLOAT) {
    PyObject *boxed_float = stack_pointer[-(1 + oparg)];
    PyObject *unboxed_float;
    double temp = ((PyFloatObject *)boxed_float)->ob_fval;
    Py_DECREF(boxed_float);
    unboxed_float = (*(PyObject **)(&temp));
    stack_pointer[-(1 + oparg)] = unboxed_float;
    DISPATCH();
}
```

One area where this approach differs from the Multi Level Quickening paper is that instead of representing the operand stack as a C `union`, it simply uses the fact that 64-bit operating systems have 64-bit pointers. Since `double`s are also 64-bit, the instruction can write the `double`'s value into the pointer. This is undefined behavior on non-64 bit systems, and type punning is generally not recommended as compiler instruction reordering could break the code. However, this approach was taken to minimise the invasive changes to CPython's runtime. A production-ready implementation of this should preferably use C unions.

Unboxed `double` thus allows for native arithmetic `double` operations. For example, the following is an example of a native `double` addition.

```
// Retrieved from /Python/bytecodes.c
inst(BINARY_OP_ADD_FLOAT_UNBOXED, (
    left, right
    --
    sum : PyRawFloat_Type)) {
    STAT_INC(BINARY_OP, hit);
    double temp = *(double *)(&(left)) + *(double *)(&(right));
    sum = *(PyObject **)(&temp);
}
```

This is the code generated and executed at runtime:

```
// Retrieved from /Python/generated_cases.c.h
TARGET(BINARY_OP_ADD_FLOAT_UNBOXED) {
    PyObject *right = stack_pointer[-1];
    PyObject *left = stack_pointer[-2];
    PyObject *sum;
    STAT_INC(BINARY_OP, hit);
    double temp = *(double *)(&(left)) + *(double *)(&(right));
    sum = *(PyObject **)(&temp);
    STACK_SHRINK(1);
    stack_pointer[-1] = sum;
    DISPATCH();
}
```

Multiplication and subtraction counterparts were also added. There are multiple benefits to these operations. Namely, the following are eliminated over operations on `PyFloatObject`:

- Reference counting
- Type checks
- Pointer chasing
- Intermediate object allocation

To handle cases where a `PyFloatObject` is needed (such as when passing a `double` as an argument to a function call), pyLBBV reboxes the `double` into a `PyFloatObject` only when required, depending on the type context. This is implemented elegantly with a `BOX_FLOAT` instruction, which takes a `double` from the stack, allocates a new `PyFloatObject` and writes that back.

The following is the CPython instruction DSL definition of `BOX_FLOAT`. Note the type stack effect: it writes a `PyFloat_Type` to the type stack at the `oparg`'s offset.

```
// Retrieved from /Python/bytecodes.c
inst(BOX_FLOAT,(
    raw_float, unused[oparg]
    --
    boxed_float : PyFloat_Type, unused[oparg])) {
    boxed_float = PyFloat_FromDouble(*(double *)(&(raw_float)));
}
```

This is the code generated and executed at runtime:

```
// Retrieved from /Python/generated_cases.c.h
TARGET(BOX_FLOAT) {
    PyObject *raw_float = stack_pointer[-(1 + oparg)];
    PyObject *boxed_float;
    boxed_float = PyFloat_FromDouble(*(double *)(&(raw_float)));
    stack_pointer[-(1 + oparg)] = boxed_float;
    DISPATCH();
}
```

Finally, the runtime needs to handle cases where an exception may arise. During an exception, the frame needs to decrement the reference count of all items on the operand stack, as well as the locals in the locals array. Unboxed values have no reference count and thus this operation should not be done on them. As such, we store a boolean flag array corresponding to the locals in the locals array in the frame object, and update this on transitions between boxed and unboxed stores (and vice versa) to the locals array.

For example, this instruction is emitted when we see that the a boxed local is being overwritten with an unboxed value.

```
// Retrieved from /Python/bytecodes.c
inst(STORE_FAST_UNBOXED_BOXED,
(value--),
locals[oparg] = *value) {
    SETLOCAL(oparg, value);
    _PyFrame_GetUnboxedBitMask(frame)[oparg] = true;
}
```

This is the code generated and executed at runtime:

```
// Retrieved from /Python/generated_cases.c.h
TARGET(STORE_FAST_UNBOXED_BOXED) {
    PyObject *value = stack_pointer[-1];
    SETLOCAL(oparg, value);
    _PyFrame_GetUnboxedBitMask(frame)[oparg] = true;
    STACK_SHRINK(1);
    DISPATCH();
}
```

Handling unboxed values on the stack during exceptions is left as future work.

5 Results

For benchmarking, this report used a computer with an Intel Core `i7-7700HQ` processor, with 16GB of RAM running the operating system Ubuntu 22.04 LTS 64-bit. CPython and pyLBBV were compiled with `./configure --with-lto` and `make -j8`, using the standard build instructions for CPython and the modified build instructions for pyLBBV (see 8.2). This is equivalent to GCC's `-O3` flag. CPython release builds recommend enabling *Profile-guided Optimisation* (PGO) and *Link Time Optimization* (LTO). This report only enabled LTO, as PGO requires running a subset of the

CPython test suite for profiling data, and as explained in 8.4, pyLBBV cannot run the CPython test suite.

The system was tuned before benchmarking with the `pyperf` module [6] using the command `pyperf system tune` (ran as `root` user). `pyperf` is the tool used to tune the system before benchmarking in `pyperformance` [7] (CPython's standard benchmarks suite). The report uses the default parameters of `pyperf system tune`, without additional system tuning. This disables Turbo Boost (dynamic CPU frequency scaling), among many other things. For a full list of `pyperf system tune` operations, see [8].

This is the code used to benchmark unboxed float arithmetic:

```
import time

def f(a, b, loops=200_000_000):
    z = a + b
    for _ in range(loops):
        z + z + z + z + z + z + z + z + z + z + z + z + z + z + z + z

# Warmup
f(1.0 , 2.0, 64)
f(1.0 , 2.0, 64)

# Running the actual benchmark

print("Starting benchmark...")
start = time.perf_counter()
f(1.0 , 2.0)
print("Time taken is", time.perf_counter() - start, "s")
```

This report acknowledges that the code above is contrived and does not represent real-world workloads. There is however some speedup (Table 5). The speedup is a demonstration that pyLBBV's type propagation and basic optimisations are indeed working. This is the best-case scenario for pyLBBV as type information is propagated perfectly and all values are unboxed.

This report attempted to benchmark a modified version of the `nbody` workload from the `pyperformance` benchmark suite [7]. `pyperformance` is "intended to be the authoritative benchmark for all Python implementations" [7]. `pyperformance` is used by CPython to track performance [9]. However, no speedup was observed for `nbody`.

Table 5: Benchmark Results

Benchmark	CPython main /s	pyLBBV /s	Relative % speedup over CPython main
`bm_float_unboxed`	21.91	13.18	39.84
`bm_nbody` [1]	30.73	30.82	-0.29

[1] Modified from the `pyperformance` version.

This report attributes the lack of speedup in `nbody` to a few possible factors:

- pyLBBV generates more instructions at every branch.
- The original lazy basic block versioning [1] paper was a technique for JIT compilation which generates native machine code with lower execution overhead.
- Arithmetic `double` operations are "cheap" in comparison. CPython 3.12 already has Tier 1 specialised instructions for float arithmetic. These instructions re-use intermediate float values if their reference count is 1, thus eliminating most object allocation.
- `nbody` includes not just `double` arithmetic operations, but list subscripting operations and function calls. These operations require reboxing of `double`.

- pyLBBV's `PyFloat_Type` unboxing is somewhat suboptimal, and requires unnecessary reboxing and unboxing at certain places.
- pyLBBV added over 30 new instructions to CPython's interpreter, which may have increased branched mispredictions in the CPython interpreter and increased instruction dispatch overhead.

6 Project Impact

The types propagator has been submitted for consideration to the Faster CPython team at Microsoft. So far, the team's response has been very positive. We are looking to collaborate with them to get an improved version of the types propagator upstream to CPython when it requires it. **If** upstreamed, the types propagator will benefit millions of CPython users worldwide.

pyLBBV is the first (to the best of our knowledge) to implement lazy basic block versioning in CPython. pyLBBV is also the first in programming language implementation (to the best of our knowledge) to combine the technique of individual self-specialising instructions (Tier 1 instructions) with the basic block versioning infrastructure, and with unboxed interpreter arithmetic instructions (Multi Level Quickening [5]), in an interpreter context.

We suspect that we may have discovered a new type propagation algorithm that integrates with structural operational semantics. Our understanding of type propagation is that traditionally it has been done at the abstract syntax tree or control flow graph level. pyLBBV's type propagation algorithm allows composing simple type operations to express complex type rules, and adds type inference to CPython bytecode. This is done without invasive changes and builds on the existing structural operational semantics of a programming language (which is usually expressed in its microcode).

Finally, this project generates lower-level Tier 2 CPython bytecode. However, it is possible to replace the code generation step with one that targets assembly language or CPU microcode. This would allow for easy integration and testing of a plug-and-play JIT compiler.

7 Future Work

A greater source of speedups may be unboxing small integer arithmetic operations. CPython's `PyLongObject` are arbitrary precision big integers. Their values are stored in C arrays of `uint32_t` and thus require custom subroutines for arithmetic operations. It may be possible to unbox 64-bit integers into C `uint64_t` and add instructions to operate on these similar to `PyFloatObject`. These would require overflow checks after said operations to ensure Python's arbitrary precision integer semantics are respected.

Unboxing can be improved if unboxing also unboxes all variables in the same tree as the boxed `PyFloatObject`, as opposed to only unboxing the stack variable. This would result in less repeated unboxing. For this, the type propagator can be used to find the tree's nodes on the stack and locals array, and unbox them as well.

We also plan to explore sources of slowdown for pyLBBV, to achieve greater speedups for targeted workloads.

8 Repository

8.1 Viewing the Project

The repository and branch is located at https://github.com/Fidget-Spinner/cpython/tree/tier2_interpreter_no_separate_eval_no_tracer_contiguous_bbs. To see what exactly has changed versus CPython's `main` branch. Either run the `git diff` terminal command, or access the following link https://github.com/Fidget-Spinner/cpython/pull/38/files.

Most changes are located in the file `Python/tier2.c`.

8.2 Building the Project

A copy of the build instructions is located in the file `CS4215.md`.

Building CPython requires different steps depending on your platform. We have tested that our project compiles on 64-bit Windows 11 and 64-bit Ubuntu 22.04. See https://devguide.python.org/getting-started/setup-building/.

The biggest change versus the standard CPython build process is that the build system **must** have Python `>= 3.9` installed already. That Python installation **must** be aliased to the name `python3` in the terminal. The main reason for this difference is that CPython uses Python to bootstrap compilation of Python startup bytecode. However, since our runtime cannot run a large portion of the Python language, our CPython build cannot be used as a bootstrap Python. All other steps are exactly the same as CPython's documented build process.

After building, the custom Python executable can be executed with either `python.bat` on Windows, or `python` on Unix-based operating systems. During the build process, errors will arise. However, the final executable should still be produced.

8.2.1 Compile-time `#define` Flags

For easier debugging and visualisation of the type propagator and basic block generation, pyLBBV has two `#define` flags for developers to modify in `Python/tier2.c`:

```
// Retrieved from /Python/tier2.c

// Prints codegen debug messages
#define BB_DEBUG 0

// Prints typeprop debug messages
#define TYPEPROP_DEBUG 0
```

Setting any of these flags to `1` will enable these features. Note that enabling these flags requires a debug version of CPython.

8.3 Documentation

Doxygen documentation is written alongside all functions in `Python/tier2.c`. All functions are documented with explanations about their behavior.

8.4 Running Tests

After building pyLBBV, run the file `tier2_test.py` located in the root directory. A lack of `AssertionError` indicates that all the tests have passed.

pyLBBV's test suite tests for nearly all optimisations that pyLBBV performs. The following are tested:

- Backward jumps
- Loop peeling
- Double unboxing
- Guard elimination for arithmetic and containers
- Support for type polymorphism
- Tier 2 `BB_TEST_ITER` specialisation

pyLBBV's tests pass on 64-bit Windows 11 and 64-bit Ubuntu 22.04. 64-bit macOS is not tested. pyLBBV tests cover the main features of the runtime. However, due to the experimental nature of this project, and the scope of the Python language, we are unable to write tests that are as comprehensive as we would like.

pyLBBV does not pass the CPython test suite, as it does not support a large portion of the Python.

8.5 Benchmark Scripts and Results

The benchmark scripts are `bm_nbody.py` and `bm_float_unboxed.py` in the root directory. Their results and the respective commit and branch they were ran against is found in the directory `/tier2_results/`.

9 Thanks and Acknowledgements

We thank the following people:

- Associate Professor Martin Henz (Project Supervisor)
- Dr Maxime Chevalier-Boisvert
- Professor Stefan Brunthaler
- the Faster CPython team at Microsoft
- the Cinder team at Meta/Instagram

for their invaluable guidance and advice.

References

[1] Chevalier-Boisvert, M., Feeley, M.: Simple and effective type check removal through lazy basic block versioning. CoRR **abs/1411.0352** (2014) 1411.0352

[2] Tiobe index (2023). https://www.tiobe.com/tiobe-index/

[3] CPython (2023). https://github.com/python/cpython

[4] A higher level definition of the bytecode interpreter (2023). https://github.com/faster-cpython/ideas/blob/main/3.12/interpreter_definition.md

[5] Brunthaler, S.: Multi-level quickening: Ten years later. CoRR **abs/2109.02958** (2021) 2109.02958

[6] Python pyperf module (2023). https://pyperf.readthedocs.io/en/latest/

[7] The Python Performance Benchmark Suite (2023). https://pyperformance.readthedocs.io/

[8] Operations and checks of the pyperf system command (2023). https://pyperf.readthedocs.io/en/latest/system.html#operations

[9] Python Speed Center (2023). https://speed.python.org/

Appendix A Dynamic Semantics from DSL to Type Propagator Behaviour

We introduce some notation. We denote `literal` to be a typeliteral, `unknown` to be a typeliteral that has the value `NULL`, `expr` to be an expression where the only free variable (if any) is `oparg`, and $\texttt{x}^n$ to be the n-th input/output stack variable from the left. In the absence of a superscript, the position of the stack variable is assumed to not matter. Input/output will be distinguished by `"--"`. For instance, for the instruction header:

```
inst(BINARY_CHECK_INT, (
    left, right
    --
    left: <<= PyLong_Type, right: <<= PyLong_Type))
```

The `left` input/output variable can be represented as $\texttt{x}^1$ and the `right` input/output variable as $\texttt{y}^2$. Further, we denote ϵ as the "do nothing" operation. We can now write the formal specification:

$$\frac{\texttt{dst} = \texttt{"unused"}}{\texttt{"--"}\ \texttt{dst} \gg \epsilon}\ \textbf{[Unused Var]}$$

$$\frac{\texttt{src} = \texttt{dst}}{\texttt{src}^n\ \texttt{"--"}\ \texttt{dst} \gg \epsilon}\ \textbf{[Unmoved Var]}$$

$$\frac{\texttt{src} \neq \texttt{dst}}{\texttt{src}\ \texttt{"--"}\ \texttt{dst} \gg \texttt{TYPE\_OVERWRITE}(\texttt{dst}, \texttt{unknown})}\ \textbf{[New Stackvar]}$$

$$\frac{\texttt{dst} \neq \texttt{"unused"}}{\texttt{"--"}\ \texttt{dst}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"} \gg \forall i \in [0, \texttt{expr} - 1] \cap \mathbb{Z},\ \texttt{TYPE\_OVERWRITE}(\texttt{dst}[i], \texttt{unknown})}\ \textbf{[New Sized Stackvar]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{literal} \gg \texttt{TYPE\_OVERWRITE}(\texttt{dst}, \texttt{literal})}\ \textbf{[Literal Typeoverwrite]}$$

$$\frac{}{\texttt{src}\ \texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"*"}\ \texttt{src} \gg \texttt{TYPE\_OVERWRITE}(\texttt{dst}, \texttt{src})}\ \textbf{[Input Typeoverwrite]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"locals"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"} \gg \texttt{TYPE\_OVERWRITE}(\texttt{dst}, \texttt{GET\_LOCAL}(\texttt{expr}))}\ \textbf{[Locals Typeoverwrite]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"consts"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"} \gg \texttt{TYPE\_OVERWRITE}(\texttt{dst}, \texttt{GET\_CONST}(\texttt{expr}))}\ \textbf{[Consts Typeoverwrite]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"<<="}\ \texttt{literal} \gg \texttt{TYPE\_SET}(\texttt{dst}, \texttt{literal})}\ \textbf{[Literal Typeset]}$$

$$\frac{}{\texttt{src}\ \texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"<<="}\ \texttt{src} \gg \texttt{TYPE\_SET}(\texttt{dst}, \texttt{src})}\ \textbf{[Input Typeset]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"<<="}\ \texttt{"locals"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"} \gg \texttt{TYPE\_SET}(\texttt{dst}, \texttt{GET\_LOCAL}(\texttt{expr}))}\ \textbf{[Locals Typeset]}$$

$$\frac{}{\texttt{"--"}\ \texttt{dst}\ \texttt{":"}\ \texttt{"<<="}\ \texttt{"consts"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"} \gg \texttt{TYPE\_SET}(\texttt{dst}, \texttt{GET\_CONST}(\texttt{expr}))}\ \textbf{[Consts Typeset]}$$

$$\frac{m \neq n}{\texttt{x}^m, \texttt{y}^n\ \texttt{"--"}\ \texttt{a}^m\ \texttt{":"}\ \texttt{"*"}\ \texttt{y}, \texttt{b}^n\ \texttt{":"}\ \texttt{"*"}\ \texttt{x} \gg \texttt{TYPE\_SWAP}(\texttt{x}, \texttt{y})}\ \textbf{[Typeswap]}$$

$$\frac{}{\texttt{"locals"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"}\ \texttt{"="}\ \texttt{literal} \gg \texttt{TYPE\_OVERWRITE}(\texttt{GET\_LOCAL}(\texttt{expr}), \texttt{literal})}\ \textbf{[Literal Localeffect]}$$

$$\frac{}{\texttt{"locals"}\ \texttt{"["}\ \texttt{expr}\ \texttt{"]"}\ \texttt{"="}\ \texttt{"*"}\ \texttt{src} \gg \texttt{TYPE\_OVERWRITE}(\texttt{GET\_LOCAL}(\texttt{expr}), \texttt{src})}\ \textbf{[Input Localeffect]}$$

Note how pyLBBV handles type propagation when no type annotations are present. This allows pyLBBV to generate the type propagator even when only a few instructions are annotated. What is not specified above is the case when the type annotation contains a list of type operations, such as `BINARY_CHECK_FLOAT` which performs unboxing as well:

```
// Retrieved from /Python/bytecodes.c
inst(BINARY_CHECK_FLOAT, (
    left, right
    --
    left_unboxed  : {<<= PyFloat_Type, PyRawFloat_Type},
    right_unboxed : {<<= PyFloat_Type, PyRawFloat_Type})
```

In this case the type operations are performed from left to right. With these specifications we can generate the type propagator. For instance, the specifications above for `BINARY_CHECK_FLOAT` generates the following for the type propagator:

```
// Retrieved from /Python/tier2_typepropagator.c.h
TARGET(BINARY_CHECK_FLOAT) {
    TYPE_SET(
        (_Py_TYPENODE_t *)_Py_TYPENODE_MAKE_ROOT(
            (_Py_TYPENODE_t)&PyFloat_Type),
            TYPESTACK_PEEK(1), true);
    TYPE_OVERWRITE(
        (_Py_TYPENODE_t *)_Py_TYPENODE_MAKE_ROOT(
            (_Py_TYPENODE_t)&PyRawFloat_Type),
            TYPESTACK_PEEK(1), true);
    TYPE_SET(
        (_Py_TYPENODE_t *)_Py_TYPENODE_MAKE_ROOT(
            (_Py_TYPENODE_t)&PyFloat_Type),
```

```
            TYPESTACK_PEEK(2), true);
    TYPE_OVERWRITE(
        (_Py_TYPENODE_t *)_Py_TYPENODE_MAKE_ROOT(
            (_Py_TYPENODE_t)&PyRawFloat_Type),
            TYPESTACK_PEEK(2), true);
    break;
}
```

Refer to https://github.com/Fidget-Spinner/cpython/blob/tier2_interpreter_no_separate_eval_no_tracer_contiguous_bbs/Python/tier2_typepropagator.c.h for the generated code.