Multi-Level Quickening

Stefan Brunthaler
µCSRL – Munich Computer Systems Research Laboratory
CODE – National Cyber Defense Research Institute
UniBwM – Universität der Bundeswehr München

brunthaler@unibw.de

Science progresses through paradigm shifts

Kuhn's **Structure of Scientific Revolutions**

Science progresses through paradigm shifts

Kuhn's **Structure of Scientific Revolutions**

- Progress in normal science reveals anomalies.

Science progresses through paradigm shifts

Kuhn's **Structure of Scientific Revolutions**

- Progress in normal science reveals anomalies.
- Anomalies cannot be explained satisfactorily by existing paradigm.

Science progresses through paradigm shifts

Kuhn's **Structure of Scientific Revolutions**

- Progress in normal science reveals anomalies.
- Anomalies cannot be explained satisfactorily by existing paradigm.
- Paradigm shift: reexamine foundations to devise new paradigm.

Anatomy of Interpreters

```
void interpret(code_t *c, object_t **sp)
  code_t *ip= c;
  for (;;) {
    opcode_t opcode= *ip++;
    switch (opcode) {
    case ADD:
      object_t u= *sp--;
      object_t v= *sp--;
      *sp++= u+v;
      break;
    case LOAD:
      object_t o= ...;
      *sp++= o;
      break;
    default:
      printf("error");
    }
  }
}
```

Nomenclature

- Interpreter routine
- Instruction decoding
- Instruction dispatch
- Interpreter instruction implementation

Progress in Interpreter Optimization 2008 - 2012

Multi-level Quickening merely the end-product of a sequence of smaller steps

Progress in Interpreter Optimization 2008 - 2012

Multi-level Quickening merely the end-product of a sequence of smaller steps

- 2006 – 2008 Existing models of interpreters do not explain varying optimization potential

Progress in Interpreter Optimization 2008 - 2012

Multi-level Quickening merely the end-product of a sequence of smaller steps

- 2006 – 2008 Existing models of interpreters do not explain varying optimization potential
- 2008 – 2010 Efficient Inline Caching in CPython

Progress in Interpreter Optimization 2008 - 2012

Multi-level Quickening merely the end-product of a sequence of smaller steps

- 2006 – 2008 Existing models of interpreters do not explain varying optimization potential
- 2008 – 2010 Efficient Inline Caching in CPython
- 2011 – 2012 Multi-Level Quickening

2008: Existing models of interpreter optimizations are inadequate

Until 2008 existing models of interpreter optimizations as follows:

- Efficient Interpreters
- Slow Interpreters

This model is insufficiently powerful to describe the varying optimization potential of instruction dispatch based optimizations.

Conflicting evidence: Varying optimization potential of threaded code.

Threaded code **the** predominant interpreter optimization.

Reported speedups vary substantially:

2.0	Efficient interpreters
1.2	Python et al.
0.8	Tcl (sometimes)

Conflicting evidence: Varying optimization potential of threaded code.

Instruction dispatch couldn’t possibly be the only source of slowdowns.

2x	Inefficient dispatch slowdown
100x – 1000x	Inefficient interpreter slowdown

➡ Instruction dispatch should reduce this to a 50x slowdown.

Existing paradigm met conflicting evidence, presenting a classical instance of a Kuhnian scientific progress.

- Conflicting evidence revealed through progress in interpreter optimization.
- Single model of efficient interpreters could **not** explain the evidence.
- Paradigm shift needed.

Formulation of new model: High vs Low Abstraction Level Interpreters

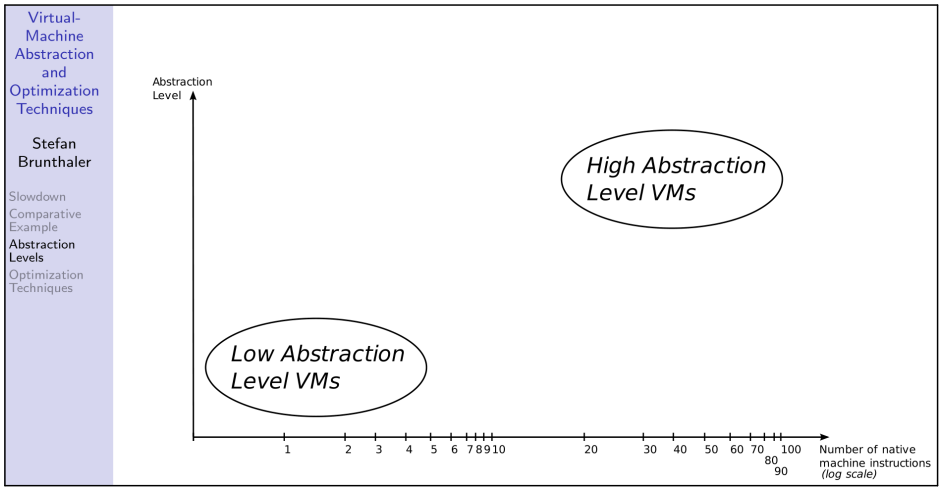
Reexamining interpreter foundations led to:

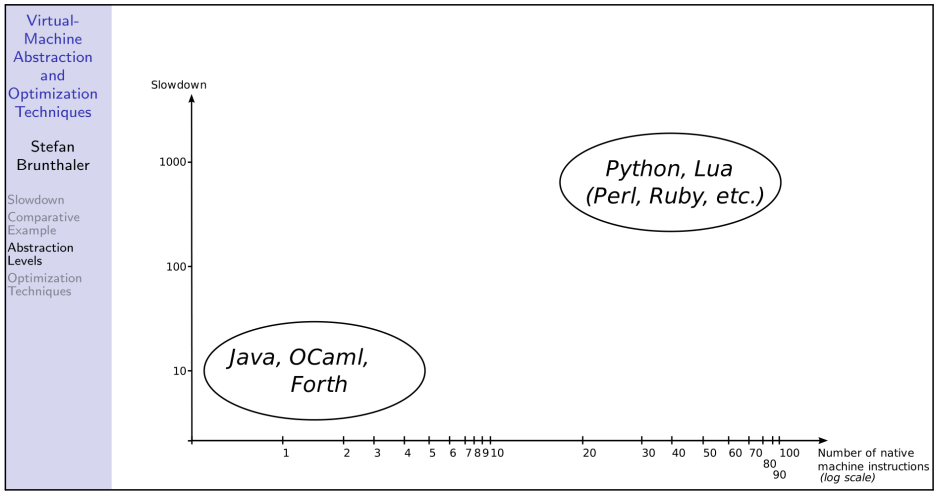
- High Abstraction-Level Interpreters Perl, Python, JavaScript.
- Low Abstraction-Level Interpreters Java, Forth, OCaml.

➡ Crucial step to consider interpreter instruction implementation complexity.

Formulation of new model: High vs Low Abstraction Level Interpreters

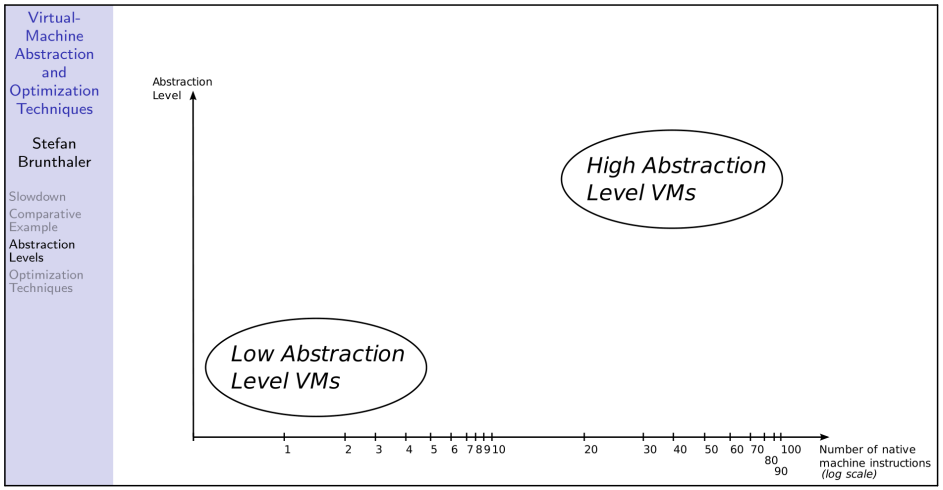
Reexamining interpreter foundations led to:

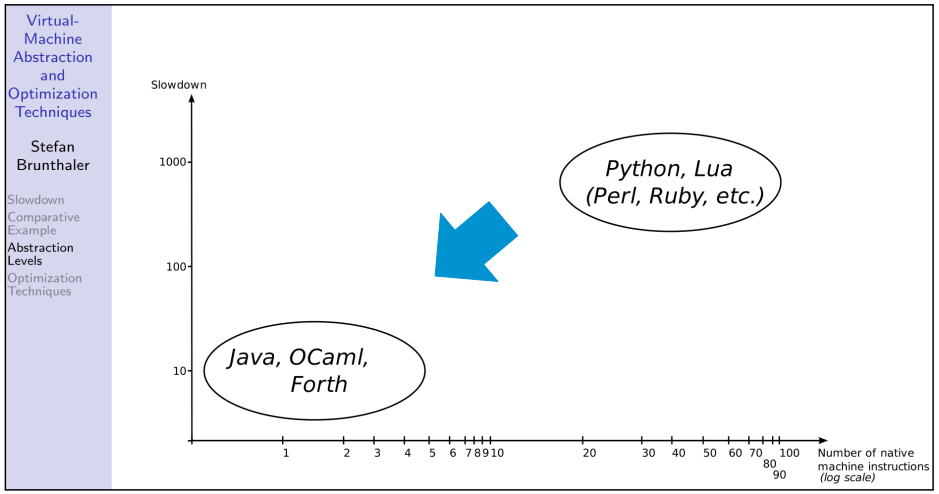




Formulation of new model: High vs Low Abstraction Level Interpreters

Reexamining interpreter foundations led to:





Low Abstraction Level Interpreters Dominated by Instruction Dispatch Costs

Interpreter instructions closely resemble native machine semantics and can, therefore, be implemented by few machine instructions.

8 – 9	Inefficient instruction dispatch
3 – 4	Efficient instruction dispatch

- Halving dispatch instructions + better utilization of branch prediction hardware.

High Abstraction Level Interpreters Dominated by Instruction Implementation Complexity

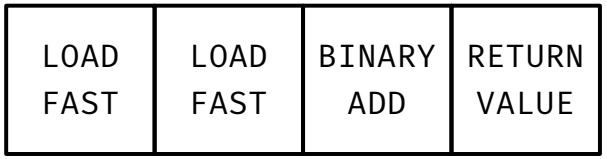
Interpreter instructions abstract native machine semantics and **cannot** be implemented by few machine instructions.

Overheads due to

- Dynamic typing
- Complex data types (unbounded range integers)
- Reference counting
- Boxed objects

Python Bytecode Interpretation and Threaded Code Performance

```
def add(a, b):
  return a + b
```

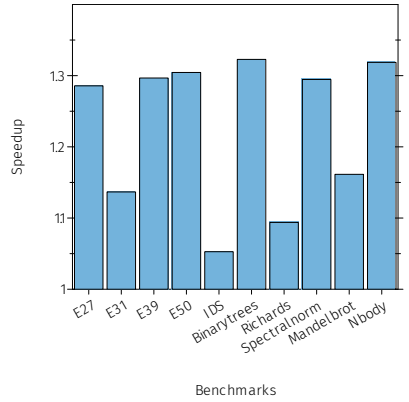



```
case BINARY_ADD:
  PyObject *right= POP();
  PyObject *left= TOP();
  PyObject *sum= PyNumber_Add(left, right);
  SET_TOP(sum);
  Py_DECREF(left);
  Py_DECREF(right);
  DISPATCH();
```

Python Bytecode Interpretation and Threaded Code Performance

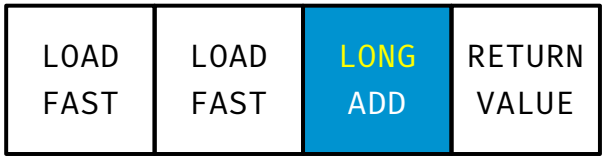
```
def add(a, b):
  return a + b
```

LOAD FAST	LOAD FAST	BINARY ADD	RETURN VALUE



Inline Caching meetings Quickening, 2009 – 2010 Idea & Performance

```
def add(a, b):
  return a + b
```

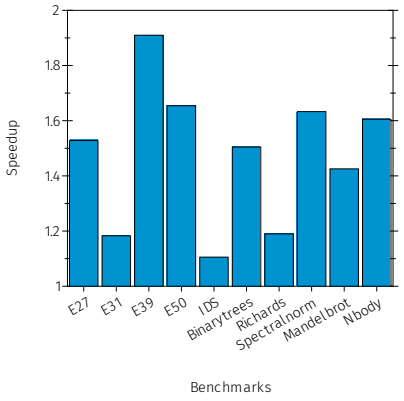



```
case LONG_ADD:
  PyObject *right= POP();
  PyObject *left= TOP();
  if (!(Py_CheckExact(left, PyLong_Type)
        && Py_CheckExact(right, PyLong_Type)))
     goto BINARY_ADD_GENERAL;
  PyObject *sum= PyLong_Type->long_add(left, right);
  SET_TOP(sum);
  Py_DECREF(left);
  Py_DECREF(right);
  DISPATCH();
```

Inline Caching meetings Quickening, 2009 – 2010
Idea & Performance

```
def add(a, b):
  return a + b
```

LOAD FAST	LOAD FAST	LONG ADD	RETURN VALUE



Multi-Level Quickening, 2010 – 2012, pt. I
Idea & Performance

```
def add(a, b):
  return a + b
```


```
case LONG_LOAD:
  PyObject *x= ...;
  if (!(Py_CheckExact(x, PyLong_Type))
     deopt();
  uint64_t nm_x= unbox(x)
  PUSH(nm_x);
  DISPATCH();
```

Multi-Level Quickening, 2010 – 2012, pt. I
Idea & Performance

```
def add(a, b):
  return a + b
```


```
case NAMA_ADD:
  uint64_t x= (uint64_t) POP();
  uint64_t y= (uint64_t) TOP();
  SET_TOP((uint64_t) x+y);
  DISPATCH();
```

Multi-Level Quickening, 2010 – 2012, pt. I
Idea & Performance

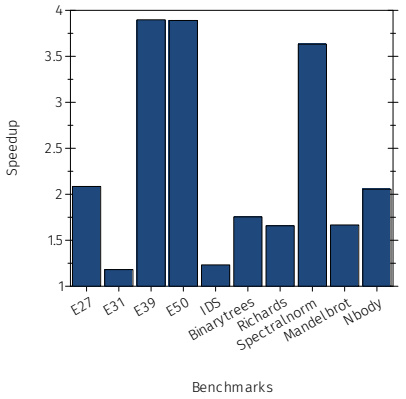
```
def add(a, b):
  return a + b
```


```
case LONG_RETURN:
  uint64_t x= (uint64_t) POP();
  ret_val= box(x, PyLong_Type)
  goto exiting;
```

Multi-Level Quickening, 2010 – 2012, pt. I
Idea & Performance

```
def add(a, b):
  return a + b
```

LONG LOAD	LONG LOAD	NAMA ADD	LONG RETURN



Multi-Level Quickening, 2010 – 2012, pt. I
Idea & Performance

```
def add(a, b):
  return a + b
```

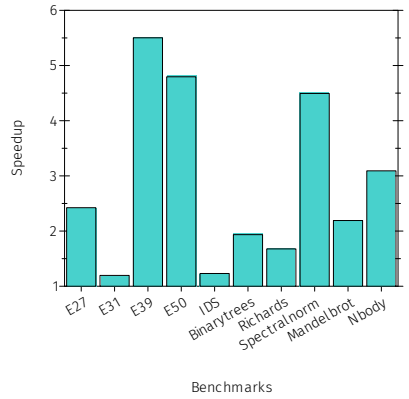
Now, instruction dispatch becomes important again!

- Concatenate sequences delimited by native-machine types
- Rewrite first member of superinstructions.

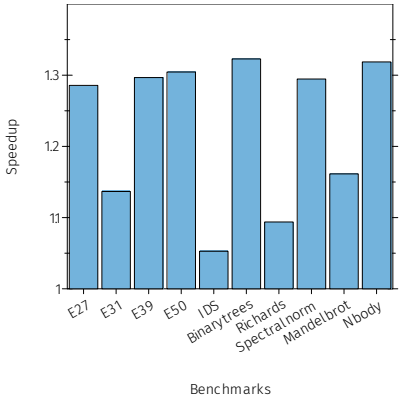
Multi-Level Quickening, 2010 – 2012, pt. II
Idea & Performance

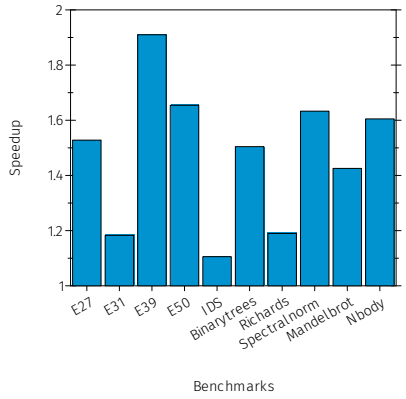
```
def add(a, b):
  return a + b
```

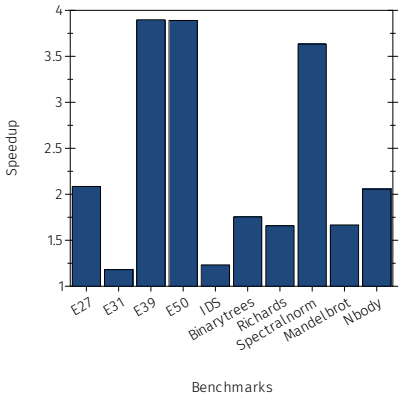
SI	LONG	NAMA	LONG
123	LOAD	ADD	RETURN

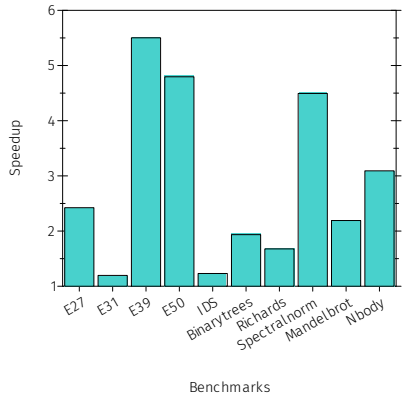


Performance Increase by up to 5.5x









A word of advice on benchmarking

MLQ performance evaluation analyzed benchmarks to understand their performance potential.

Often, programs are **not** constrained by the interpreter, so no amount of optimization will turn out beneficial.

A word of advice on benchmarking

Analyze how much time was spent in the interpreter (T_{int}) and use the corollary to Amdahl's law:

$$\text{Max. Speedup} = \frac{1}{1-T_{\text{int}}}$$

MLQ is not only limited to numbers and scalar operations

Besides supporting all numbers, MLQ also considered:

- attribute reads/writes for dictionaries and lists
- subscript reads/writes for dictionaries and lists with numeric and unicode types.
  - **`x[0]`** or **`x[10]= ...`**

Where to go from here?

Future directions for MLQ-style interpreters

Hybrid ISA interpreters

Unite the space-efficiency of stack-based ISAs with reduced dispatch costs of register-based ISAs.

➡ Additional speedup about 30pct

Future directions for MLQ-style interpreters

Feature Adaptive Interpreters

At present, most interpreters analyze programs and build in the most frequently occurring case.

Allows interpreters to specialize for *all* special cases.

➡ relevant for functional and logic PLs

Future directions for MLQ-style interpreters

Cross native-machine library MLQ

Optimization boundary usually single native-machine *object*

MLQ could provide optimization API that crosses object boundary:

- Generic abstract interpreter over abstract types.
- Library, such as numpy, registers optimized code with the MLQ-API

Future directions for MLQ-style interpreters

Generic Hidden Classes via MLQ

Well-known optimization in JIT compilers. To the best of my knowledge not used in interpreters right now.

MLQ-style interpretation:

- Adjacent memory with n slots for each field.
- Additional MLQ instructions for direct manipulation.

Future directions for MLQ-style interpreters

Generation of MLQ-style interpreters

`vmgen` and **`tiger`** interpreter generators developed at TU Wien.

MLQ relies heavily on the idea of **derivatives**, which lend themselves well to generation. To drive generation, one would need to specify:

- Numerical tower with conversions and operations.
- Mapping operations to enable boxing/unboxing.

How to think about MLQ?

Interesting Philosophical Observations of Interpreters and their Optimization

Information Theoretical Perspective of Interpreter Optimizations.

Thoughts concerning the limits of interpreters vs JIT compilers.

Thoughts on MLQ for virtual machines.

Information Theory for Regular Interpreters

$$i \xrightarrow{\text{next}} j$$

Instruction dispatch transitions from instruction i to j. The next relation merely indicates the sequence of instructions in the interpreted program P.

Information Theory for Interpreters with Inline Caching

$$i \xrightarrow{\text{next}_\tau} j$$

Instruction dispatch transitions from instruction i to j. The next relation merely indicates the sequence of instructions, and **expected types** τ.

Information Theory for Interpreters with MLQ

$$i \xrightarrow{\mathrm{next}_{\tau,\rho}} j$$

Instruction dispatch transitions from instruction i to j. The next relation merely indicates the sequence of instructions, expected types τ, and expected data representation ρ.

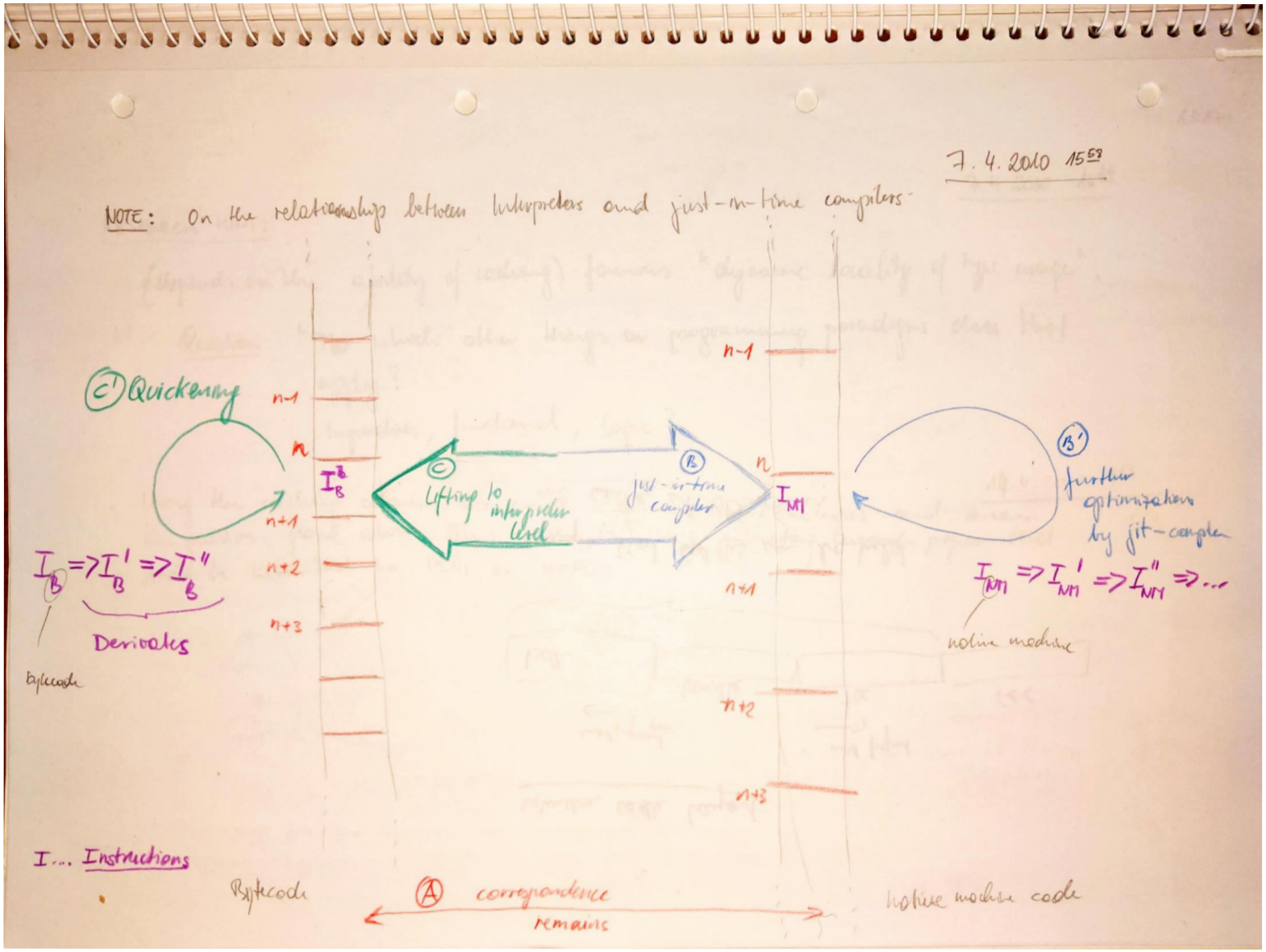
Information Theoretical Perspective

$$i \xrightarrow{\text{next}} j$$

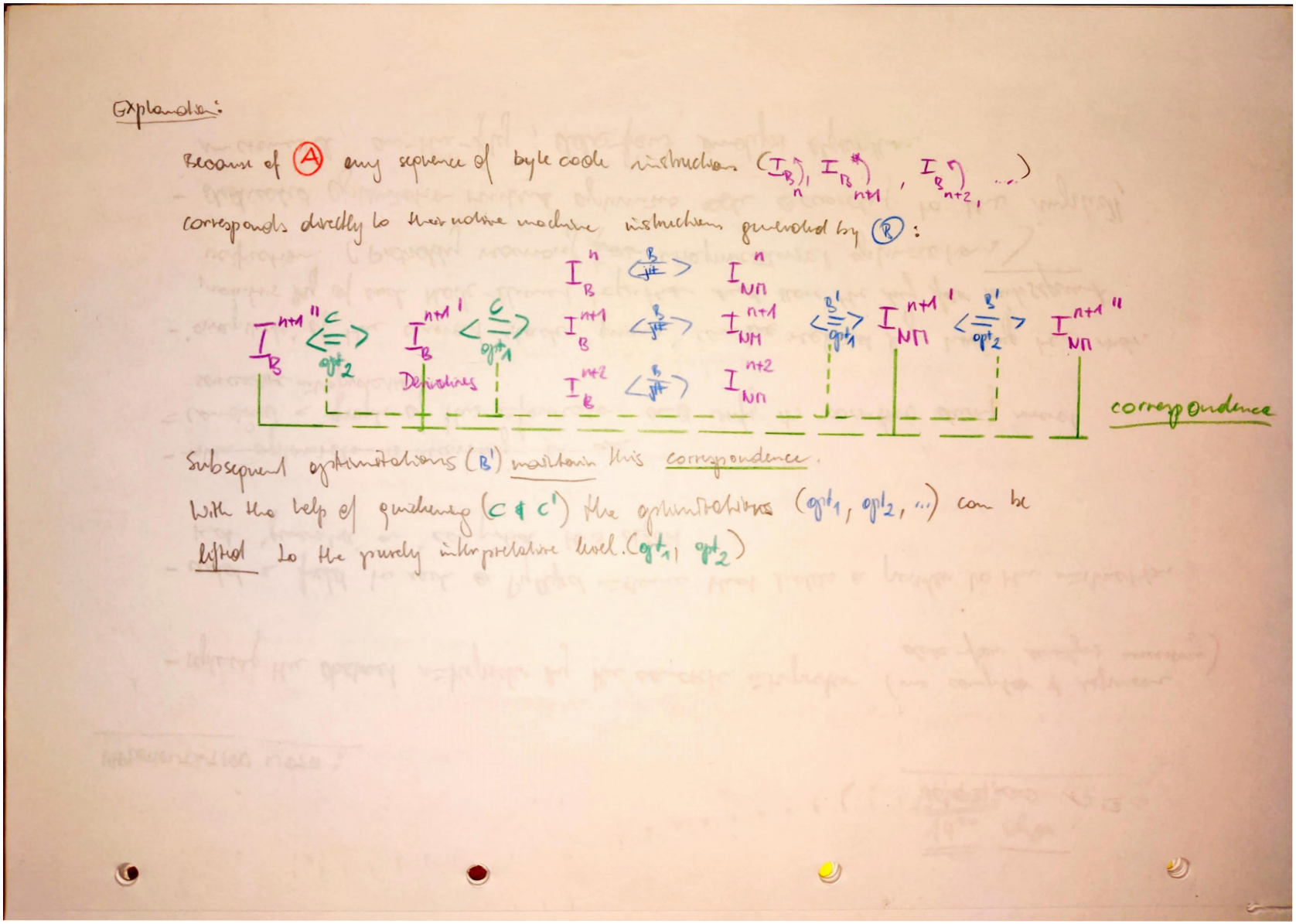
$$i \xrightarrow{\text{next}_\tau} j$$

$$i \xrightarrow{\text{next}_{\tau,\rho}} j$$

Limits for interpreters compared to JIT compilers



Limits for interpreters compared to JIT compilers



There are no theoretical limits for interpreters when compared to JIT compilers

An interpreter can always do everything a JIT compiler can do.

Since optimized interpreter instructions are being compiled by an ahead-of-time compiler, the resulting code may even be *more* optimized when compared to a JIT compiler.

Constraints and practical considerations for MLQ-style Interpreters

Maximum number of interpreter instructions

Anecdotal evidence puts the maximum number of interpreter instructions at about 2,000. This limit puts some constraints on how many derivatives an interpreter can realistically accommodate.

Constraints and practical considerations for MLQ-style Interpreters

Speculation on Derivative Utility

Derivatives are put into the interpreter source code and are, therefore, fixed at interpreter compiler-time.

- Interpreter instruction set **fixed** for subsequent execution of programs.

Careful consideration and management of expectations needed to ensure that the utility of optimized derivatives is estimated properly.

Thoughts on MLQ for virtual machines

MLQ succeeds because it **extends** CPython's instruction set.

Whereas CPython's instruction set is untyped/type-generic, MLQ adds implicit typing and data representation information, making the instruction set type-specific.

Information **internal** to the virtual machine is externalized.

Contributions

- Advanced type feedback using a purely interpretative setup.
  - ease of implementation
  - portability
  - correctness formalized & verified (CPP'21)
  - security, increasingly important
- Native-machine-type based superinstructions.
- Performance-Analysis driven Benchmark Selection and Understanding

Contributions

- Advanced type feedback using a purely interpretative setup.
  - ease of implementation
  - portability
  - correctnes ed (CPP'21)
  - security, ir
- Native-machine-type based superinstructions.
- Performance-Analysis driven Benchmark Selection and Understanding