

AMORTISED OPTIMISATION OF NON-FUNCTIONAL PROPERTIES IN PRODUCTION ENVIRONMENT

Shin Yoo (shin.yoo@kaist.ac.kr)

Korea Advanced Institute of Science and Technology

GENETIC IMPROVEMENT

Optimising Existing Software with Genetic Programming

William B. Langdon and Mark Harman

Abstract—We show genetic improvement of programs (GIP) can scale by evolving increased performance in a widely-used and highly complex 50 000 line system. GISMOE found code that is 70 times faster (on average) and yet is at least as good functionally. Indeed it even gives a small semantic gain.

Index Terms—automatic software re-engineering, SBSE, genetic programming, Bowtie2^{GP}, multiple objective exploration

I. INTRODUCTION

GENETIC improvement [1; 2; 3; 4] is the process of automatically improving a system's behaviour using genetic programming. Starting from a human written system, genetic improvement tries to evolve it so that it is better with respect to given criteria. The criteria for improvement are typically non-functional properties of the system, such as execution time and power consumption, though many others are possible [1; 4]. The functional properties of the evolved system are usually required to mimic as faithfully as possible those of the original system. However we show that it may also be possible to improve the program's outputs.

In order to check that the original's semantics are not disturbed, the genetic improvement process relies on a set of test cases, obtained from running the original system. Notice we can always do this [5]. Even where the existing

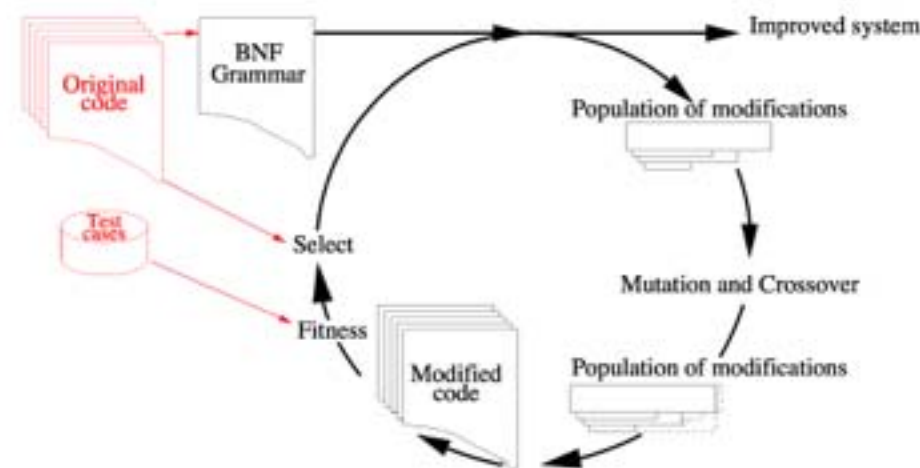


Fig. 1. Major components of GISMOE approach. Left: system to be improved and its test suite. Right: genetic programming optimises modifications which originate from a grammar that describes the original system line by line. Each generation mutation and crossover create new modifications. Each modification's fitness is evaluated by applying it to the grammar and then reversing the grammar to get a new variant of the system. Each modified system is tested on a randomised subset of the test suite and its answers and resource consumption compared to that of the original system. Modifications responsible for better systems procreate into the next generation.

currently familiar (yet time-consuming, tedious and expensive) method. Ultimately, genetic improvement looks forward to a world in which our successors regard human *programmers* as a 'quaint anachronism of the past' in much the same way that

GENETIC IMPROVEMENT

Genetic Improvement 2015

A Workshop at GECCO 2015 on the Genetic Improvement of Software

[HOME](#)[PHOTOS](#)[SLIDES](#)[PROGRAMME](#)[PAPERS](#)[ORGANISERS](#)[PROGRAM COMMITTEE](#)[SPONSORS](#)[FAQ](#)

GI 2015



The First International Genetic Improvement Workshop (GI-2015) was held in Madrid, Spain, during the Genetic and Evolutionary Computation Conference (GECCO-2015), July 11-15, 2015.

Journal Special Issue

Tweets

[Follow](#)



Nic McPhee
@NicMcPhee

1 Aug

This was a very cool workshop; thanks to all involved. [#gecco2015](#) [#geccogi2015](#)
[twitter.com/GI_2015/status...](#)

Retweeted by GI 2015

Expand



GI 2015
@GI_2015

31 Jul

Slides from half of the presenters at GI 2015 now available online!
[geneticimprovement2015.com/?page_id=28](#)

GENETIC IMPROVEMENT

Embedding Adaptivity in Software Systems using the ECSELR framework *

Kwaku Yeboah-Antwi
INRIA
Rennes, France
kwaku.yeboah-antwi@inria.fr

Benoit Baudry
INRIA
Rennes, France
benoit.baudry@inria.fr

ABSTRACT

We present, ECSELR, an ecologically-inspired approach to software evolution that allows for environmentally driven evolution in extant software systems at runtime without relying on any offline components or management. ECSELR embeds adaption and evolution inside the target software system allowing such systems to transform themselves via darwinian evolutionary mechanisms and adapt in a self contained manner. This allows such software system to benefit from the useful byproducts of evolution like adaptivity, bio-diversity without having to worry about problems involved in engineering and maintaining such properties. ECSELR allows the software systems to address changing environments at runtime enabling benefits like mitigation of attacks, memory-optimization among others while avoiding time-consuming and costly maintenance and downtime. ECSELR differs from existing work in that, 1) adaption is em-

Keywords

Software Evolution, Diversification, Software Slimming, Search Based Software Engineering, Genetic Improvement, Self Adaptive Software Engineering

1. INTRODUCTION

The field of software engineering has been straining to deal with the exponential growth of software systems. As computer systems get more and more complex, maintaining them requires a huge amount of human effort. Software Engineering may soon hit a "*complexity wall*" where our efforts will not be able to scale up to highly complex software systems [2, 6, 14]. This has led to the rise in the amount of research being done in automating software design, maintenance, with software engineers increasingly seeking to design software systems that autonomously evolve and adapt. Designing such self-optimising adaptive systems has been

GENETIC IMPROVEMENT

locoGP: Improving Performance by Genetic Programming Java Source Code

Brendan Cody-Kenny, Edgar Galván-López, Stephen Barrett
School of Computer Science & Statistics, Trinity College Dublin
{codykenb, edgar.galvan, stephen.barrett}@scss.tcd.ie

ABSTRACT

We present locoGP, a Genetic Programming (GP) system written in Java for evolving Java source code. locoGP was designed to improve the performance of programs as measured in the number of operations executed. Variable test cases are used to maintain functional correctness during evolution. The operation of locoGP is demonstrated on a number of typically constructed “off-the-shelf” hand-written implementations of sort and prefix-code programs. locoGP was able to find improvement opportunities in all test problems.

Categories and Subject Descriptors

I.2 [ARTIFICIAL INTELLIGENCE]: Automatic Programming; D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

own language (as defined by the primitives chosen), parser and interpreter. Interpretation and evaluation of programs can be achieved in a small amount of code by choosing a set of primitive functions specifically for a problem. Fortunately, there exist comprehensive libraries for parsing, compiling and interpreting more general Java programs upon which locoGP relies heavily.

In locoGP, the primitive set is defined by Java language elements which exist in the program to be improved and may include statements, expressions, variable names or operators. Source code is modified in an Abstract Syntax Tree (AST) representation which specifies the typing of nodes and structure of a program in the Java language.

Performance is measured by counting the number of instructions taken to execute a program. Program results are measured for correctness with a problem-specific function by

GENETIC IMPROVEMENT

Genetic Programming and Evolvable Machines

ISSN: 1389-2576 (Print) 1573-7632 (Online)

Description

Methods for artificial evolution of active components are rapidly developing branches of adaptive computation and adaptive engineering. They entail the development, evaluation and application of methods that mirror the process of neo-Darwinian evolution. Genetic Programming and Evolvable Machines reports innovative and significant progress in automatic evolution of software and hardware. It features both theoretical and appli ... [show all](#)

16	61	397	10	2000-2015
Volumes	Issues	Articles	Open Access	Available between

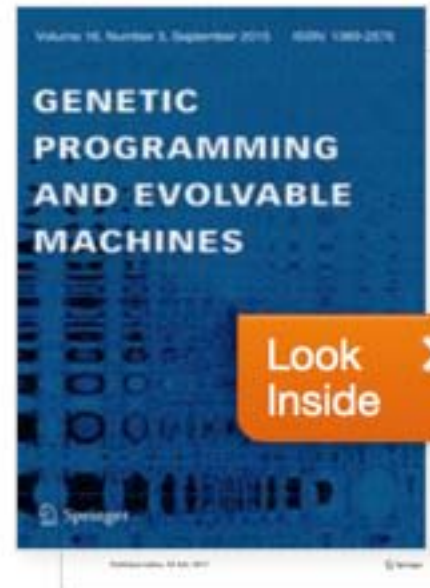
Find your Volume or Issue



Browse all Content

[Browse Volumes & Issues](#)

[View Open Access Articles](#)



Other actions

- [» Register for Journal Updates](#)
- [» About This Journal](#)

Share



OFFLINE IMPROVEMENT

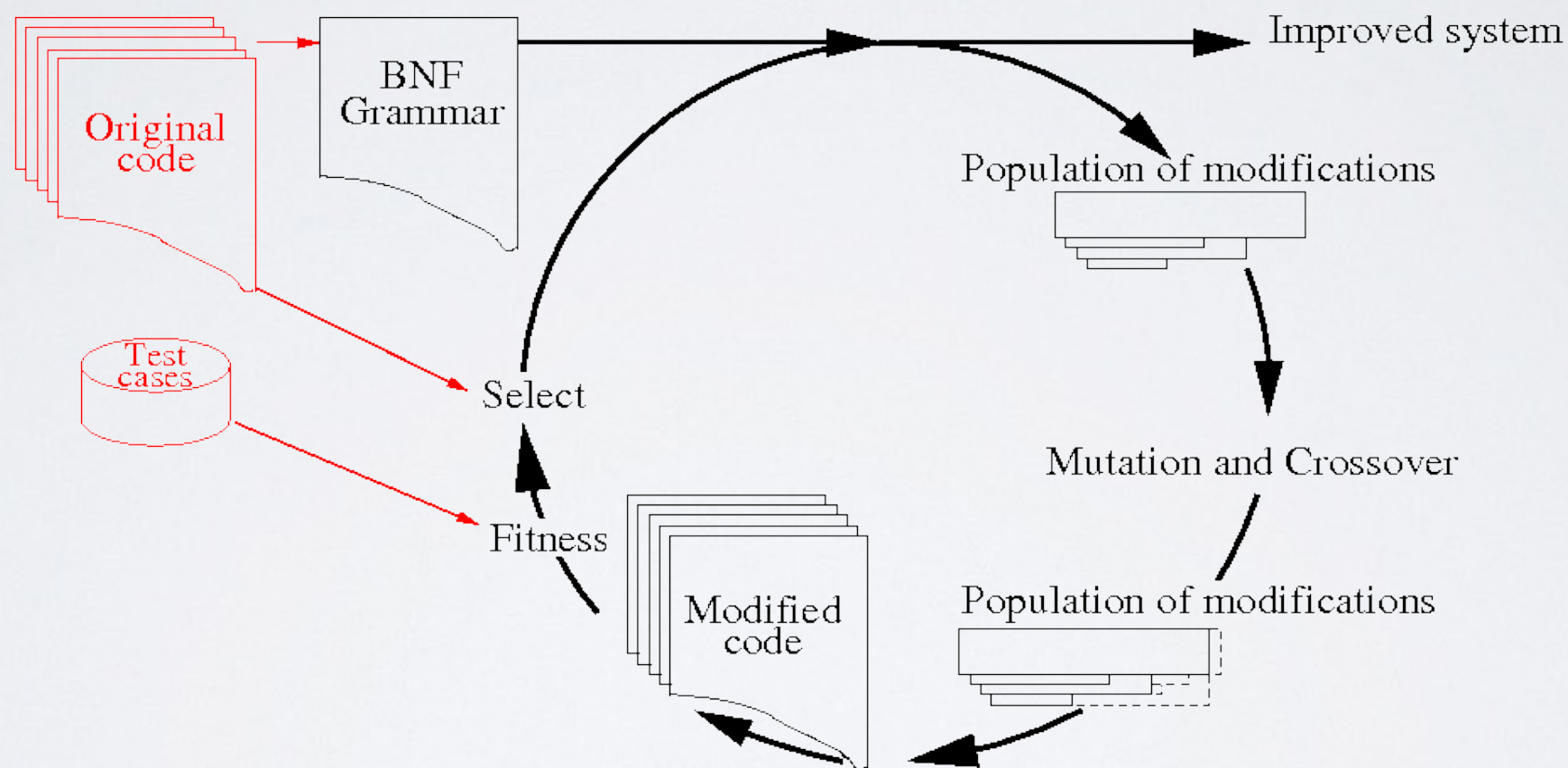


Fig. 1. GP improvement of MiniSAT.

OFFLINE IMPROVEMENT

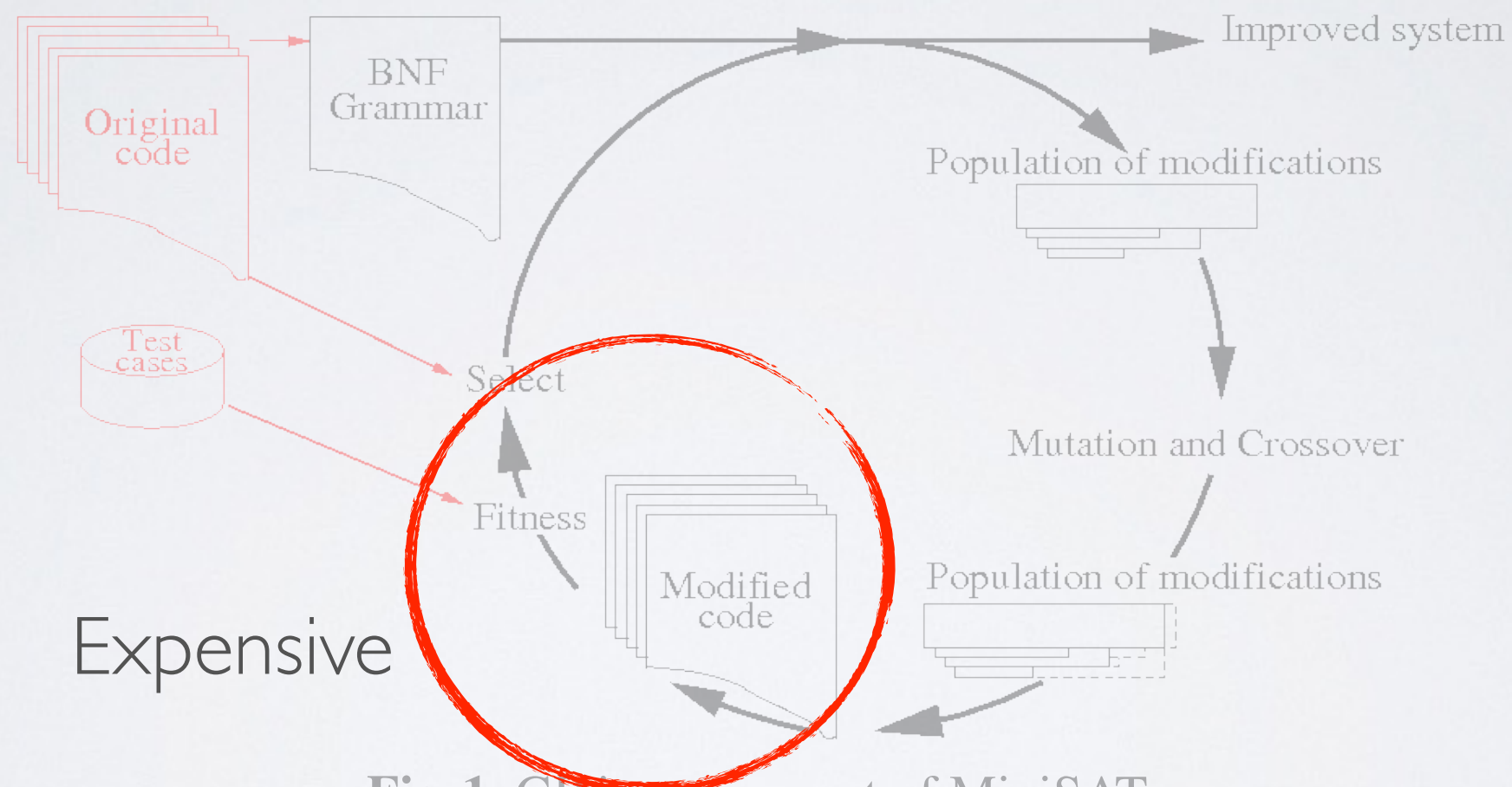
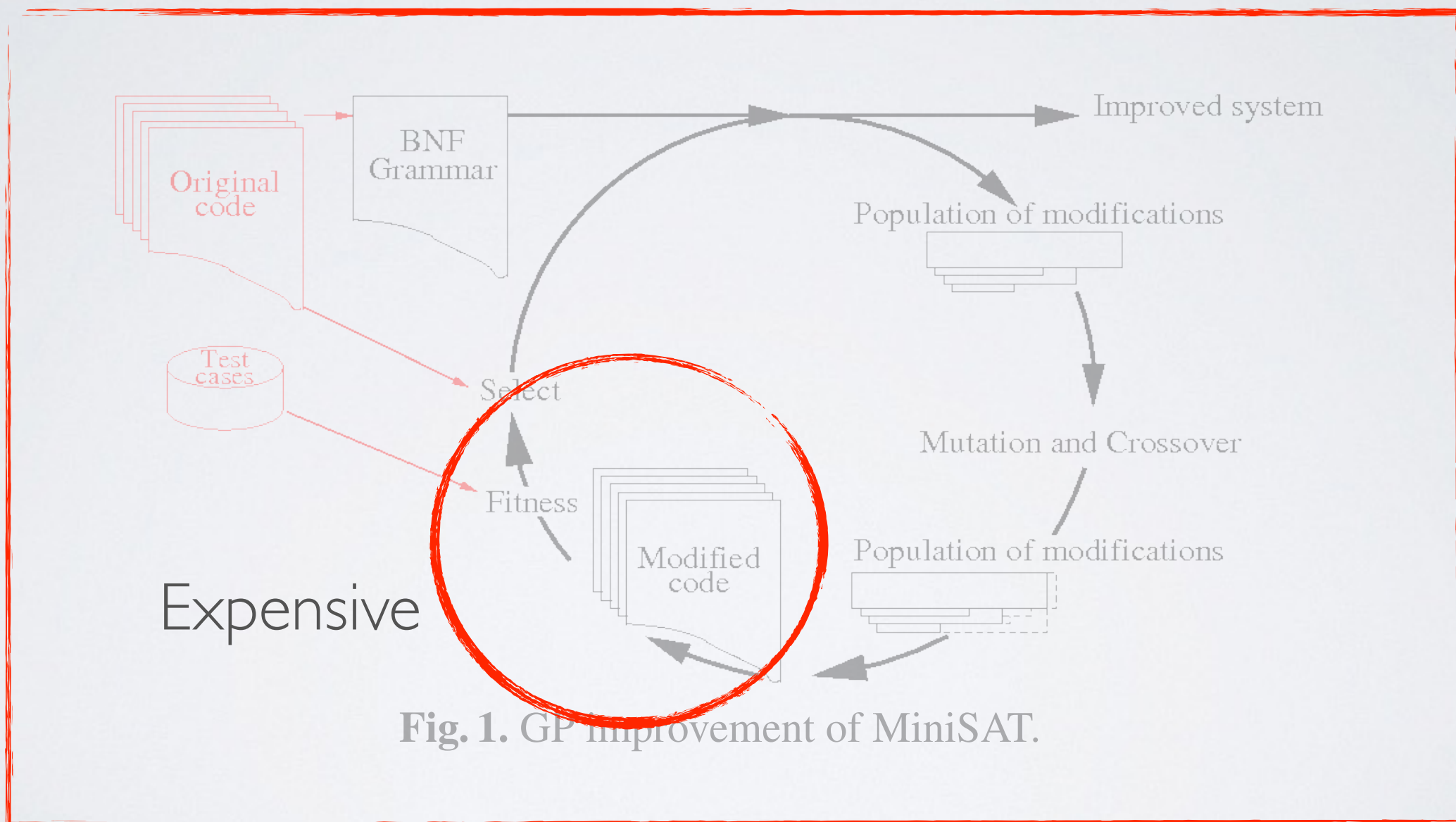


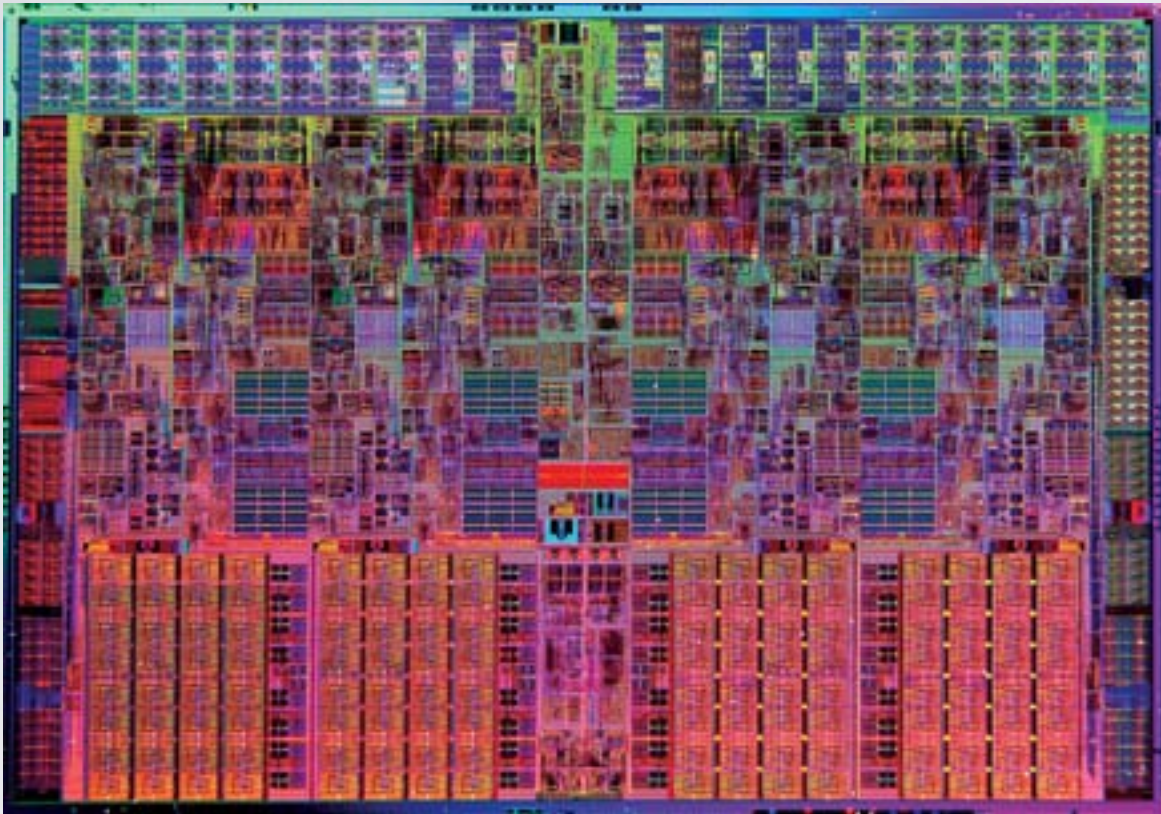
Fig. 1. GP improvement of MiniSAT.

OFFLINE IMPROVEMENT

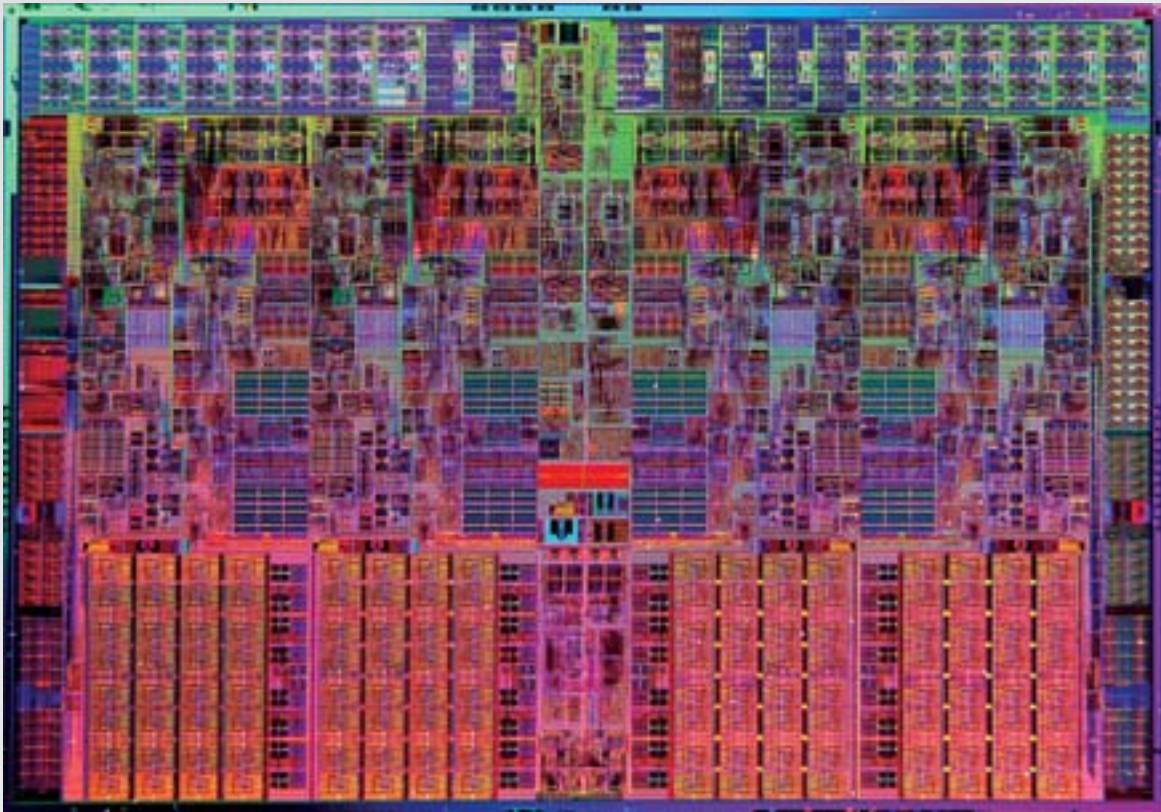


Tied to offline environment

ENVIRONMENTAL FACTORS

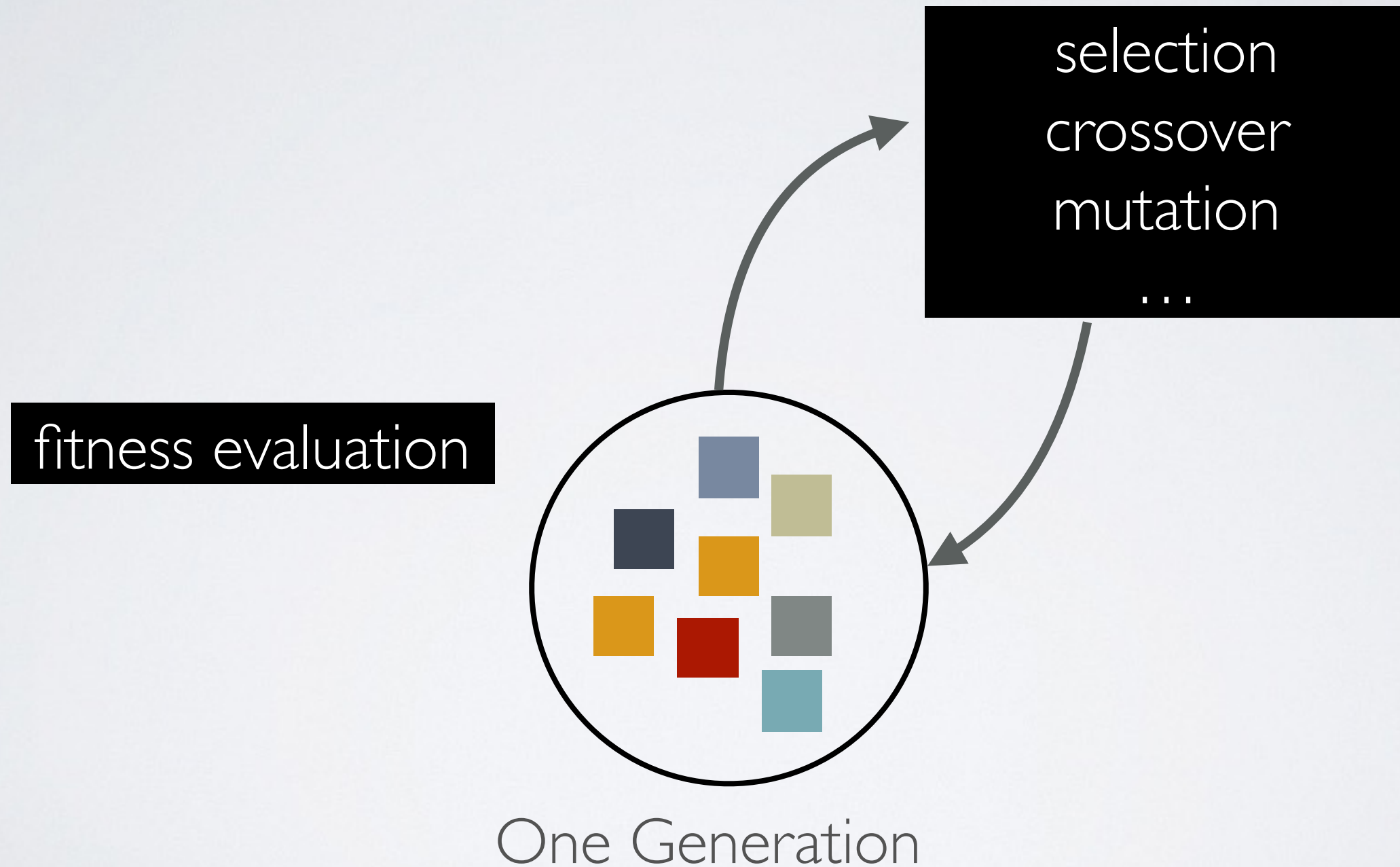


ENVIRONMENTAL FACTORS



We cannot anticipate the environment that the software will be executed; hence it is hard to optimise for it.

OFFLINE OPTIMISATION



AMORTISED OPTIMISATION

selection
crossover
mutation

...



AMORTISED OPTIMISATION



AMORTISED OPTIMISATION



AMORTISED OPTIMISATION



AMORTISED OPTIMISATION



AMORTISED OPTIMISATION



Optimisation executed in micro-steps,
each in-situ execution as a single fitness evaluation

AMORTISED OPTIMISATION

AMORTISED OPTIMISATION



*Genetic Improvement,
out in the wild!*

AMORTISED OPTIMISATION



Budget Controlled
(will stop when run out)

*Genetic Improvement,
out in the wild!*

AMORTISED OPTIMISATION



*Genetic Improvement,
out in the wild!*

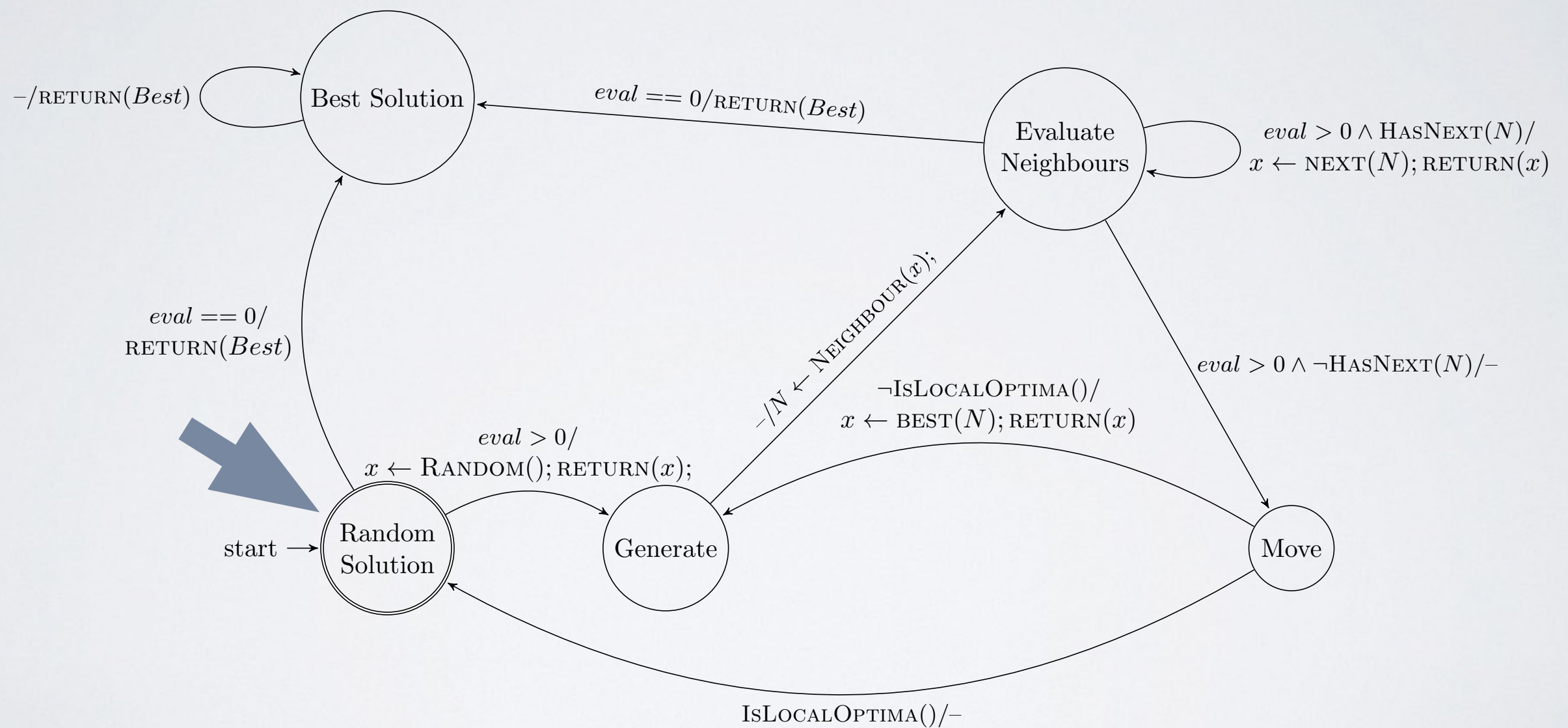


Budget Controlled
(will stop when run out)



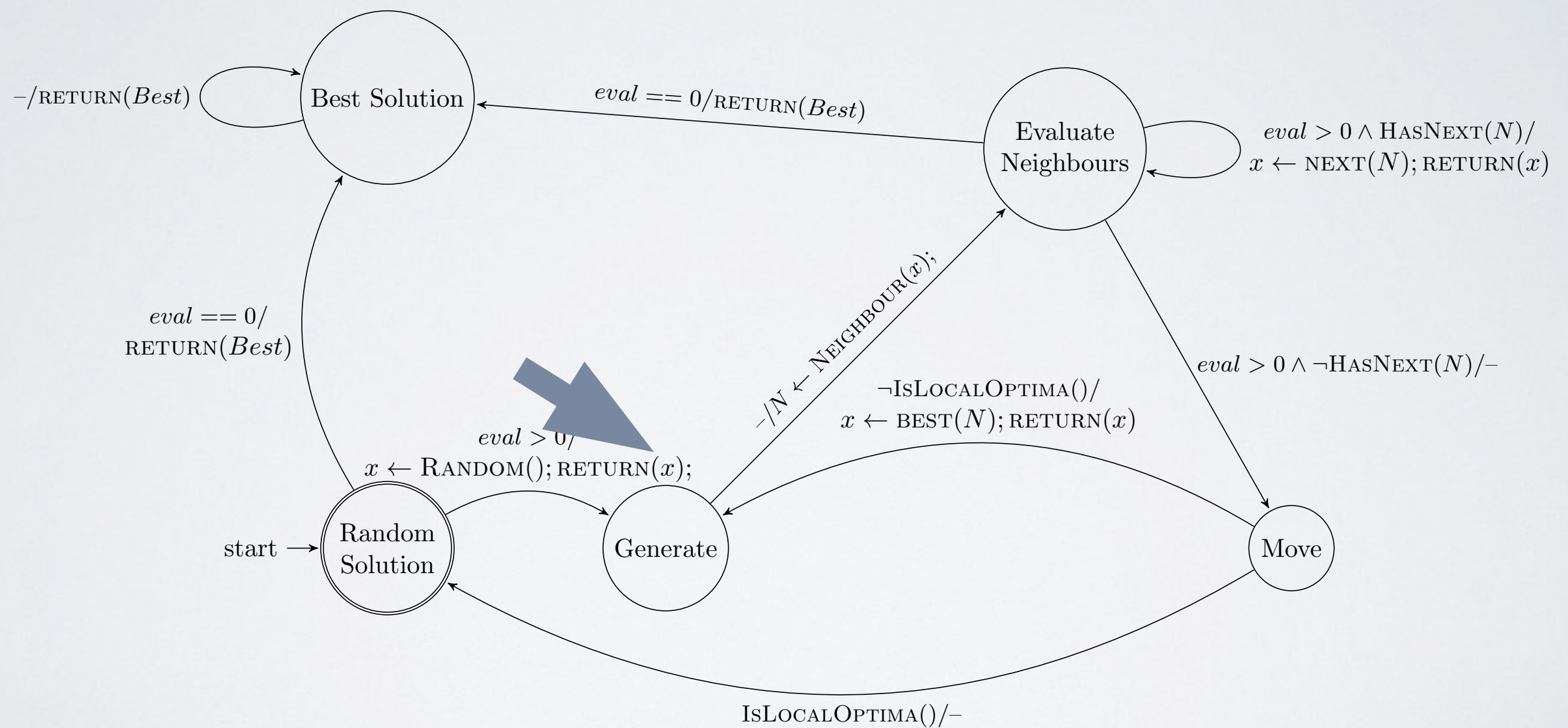
Low Overhead
(only microscopes)

STATE-BASED RECONSTRUCTION OF OPTIMISATION



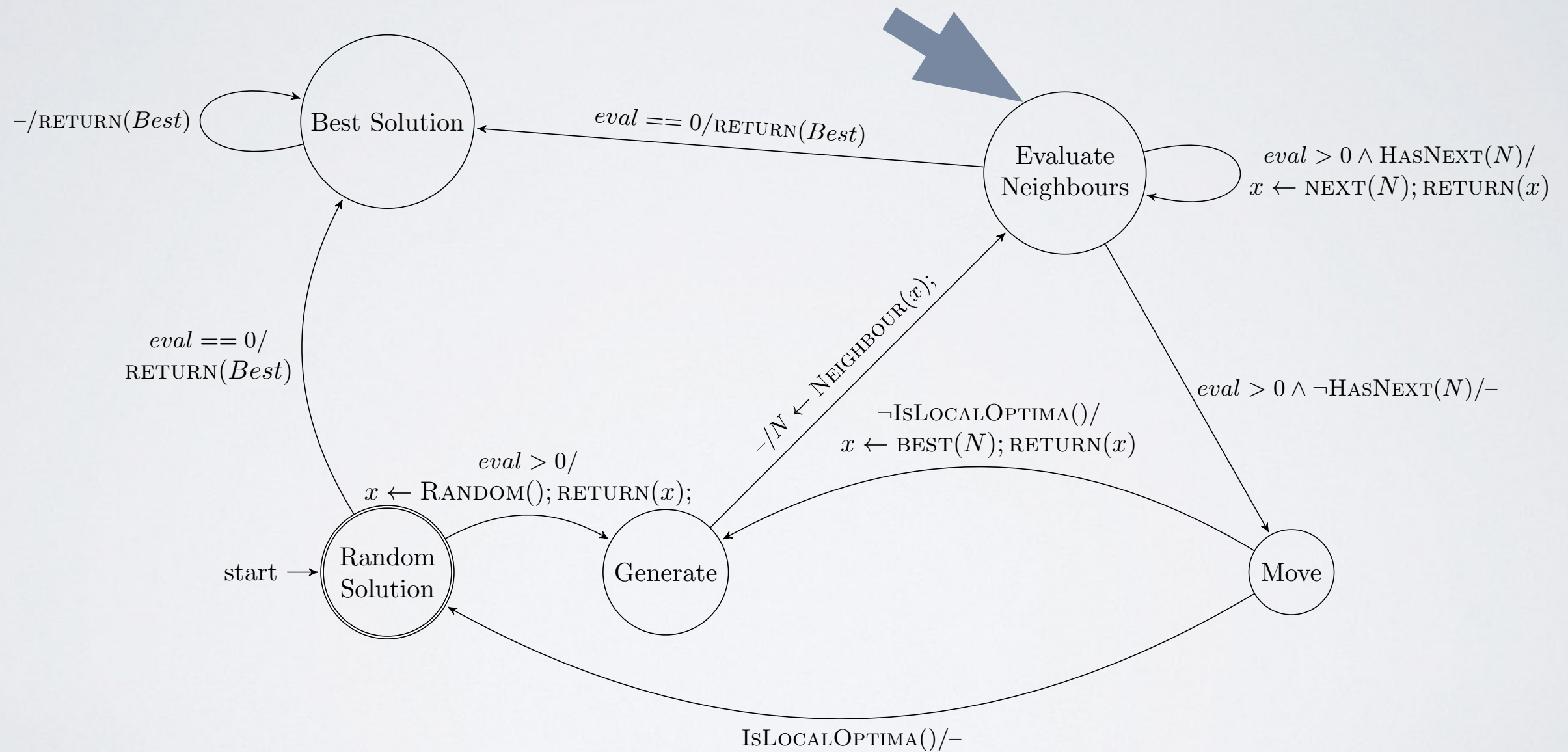
Execute transitions until you return a candidate solution to the SUMO

STATE-BASED RECONSTRUCTION OF OPTIMISATION



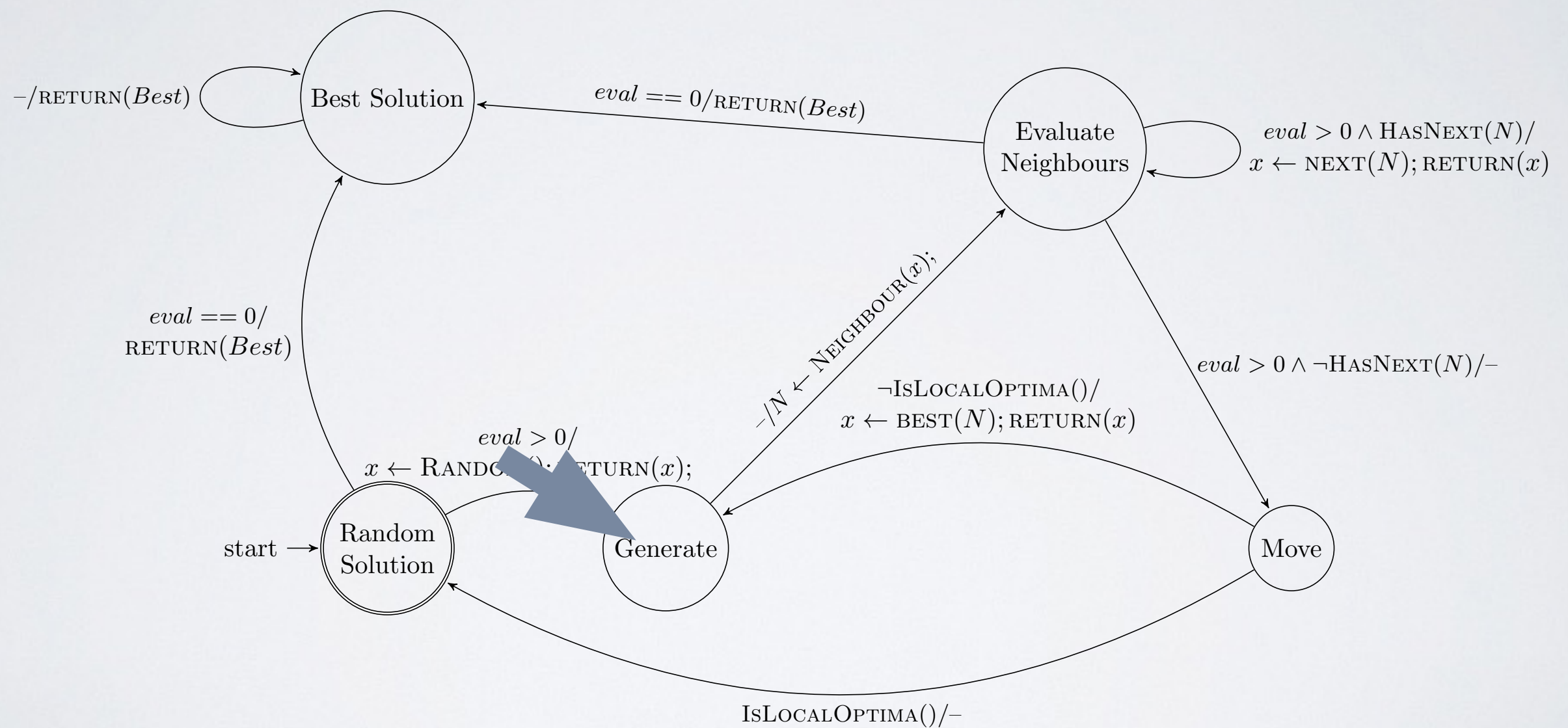
Execute transitions until you return a candidate solution to the SUMO

STATE-BASED RECONSTRUCTION OF OPTIMISATION



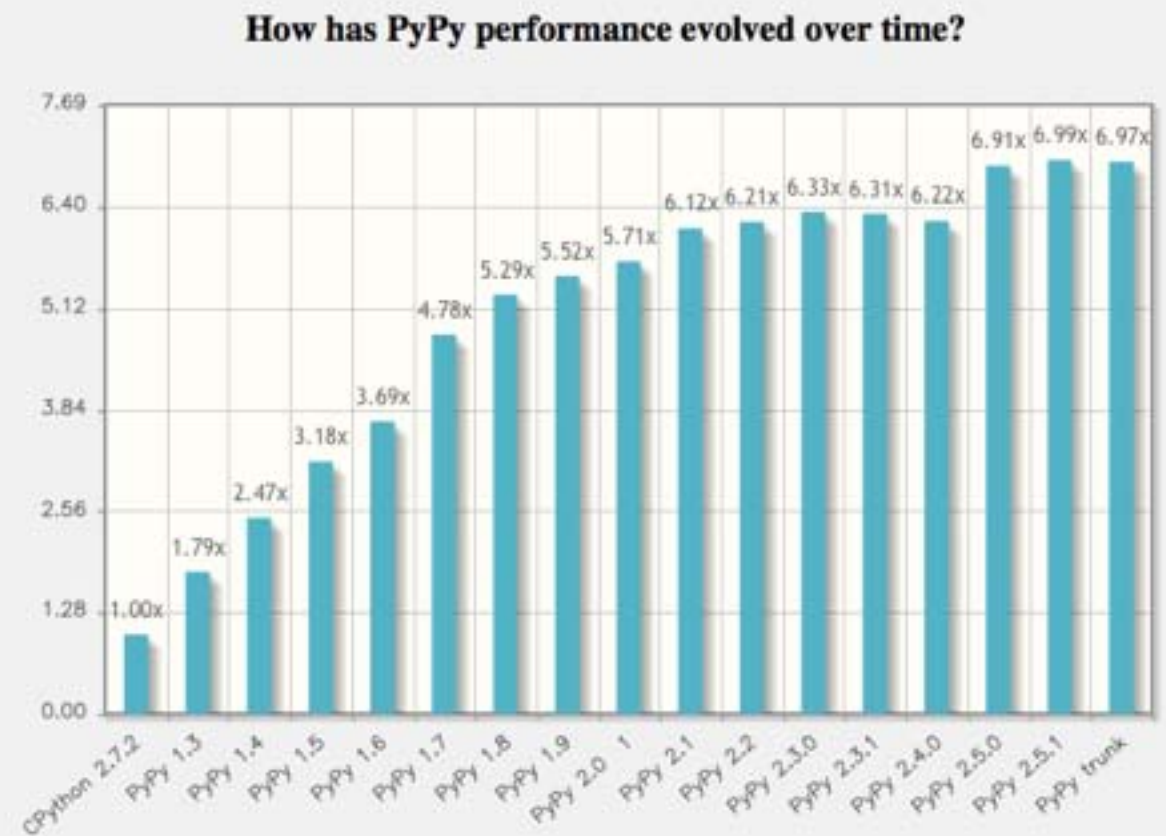
Execute transitions until you return a candidate solution to the SUMO

STATE-BASED RECONSTRUCTION OF OPTIMISATION



Execute transitions until you return a candidate solution to the SUMO

DOES IT WORK?



Plot 2: Speedup compared to CPython, using the inverse of the geometric average of normalized times, out of 20 benchmarks (see [paper](#) on why the geometric mean is better for normalized results).

We applied amortised optimisation to **pypy**,
a tracing-JIT based python implementation.

TRACING JIT

- Repeatedly executed loops and functions are marked “hot”
- Traces of hot code is translated into the native code, with guards that guarantee the correctness
- When guards fail (meaning interpretation and JIT code diverge), revert back to interpretation and recompile bridge (a sub-path that corrects the divergence)

TRACING JIT PARAMETERS

TRACING JIT PARAMETERS



When to begin
tracing?

TRACING JIT PARAMETERS



When to begin
tracing?



When to mark as
hot?

TRACING JIT PARAMETERS



When to begin tracing?

When to mark as hot?

When to compile the bridge?

PIACIN

PIACIN

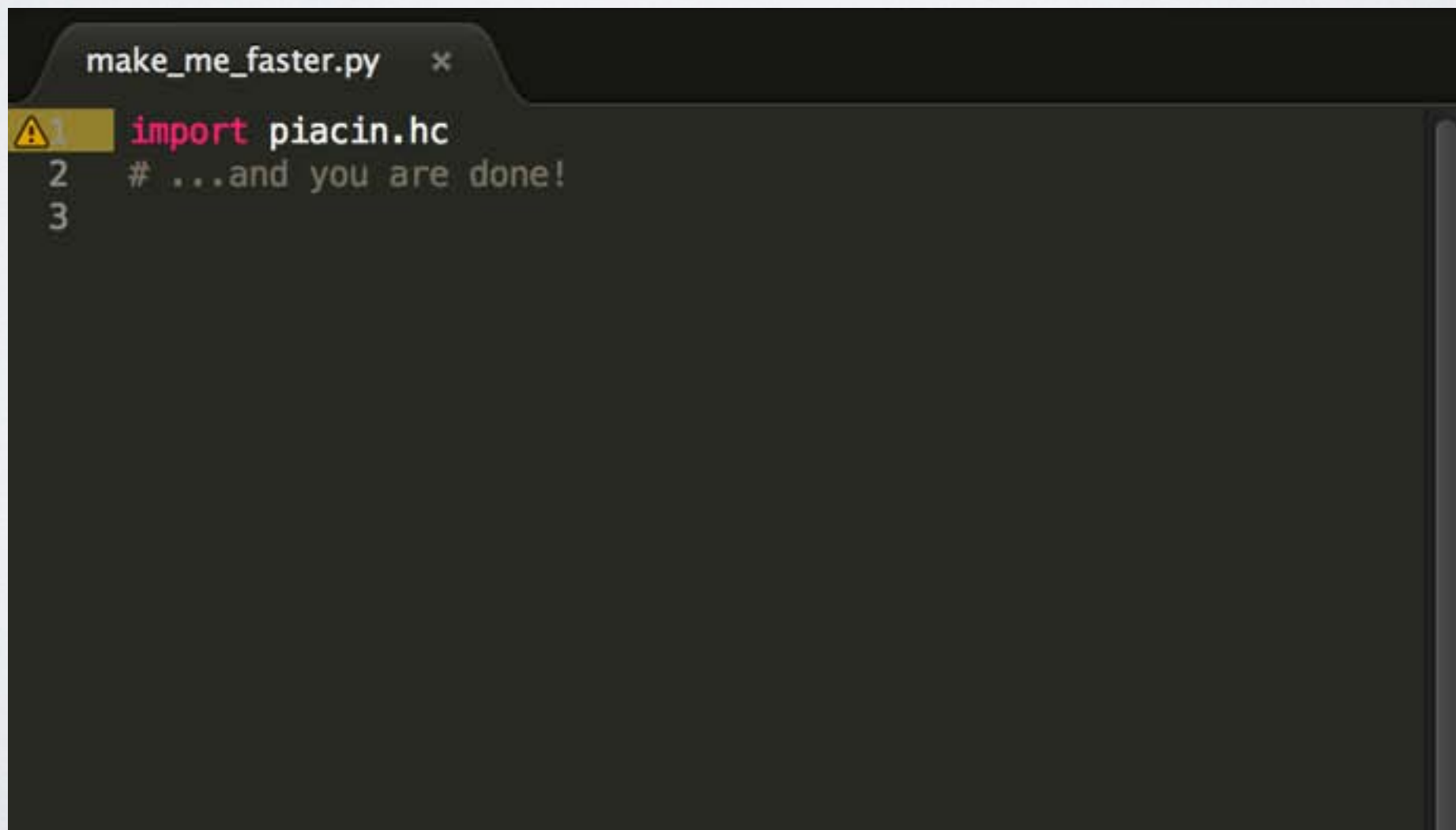
1. Install the package.

PIACIN

1. Install the package.
2. Import the package

PIACIN

1. Install the package.
2. Import the package
3. There is no step 3.



```
make_me_faster.py ×  
1 ⚠ import piacin.hc  
2 # ...and you are done!  
3
```

The screenshot shows a code editor window with a dark theme. The title bar of the window is labeled 'make_me_faster.py' with a close button. The first line of code is 'import piacin.hc', which is highlighted in yellow and has a yellow warning icon to its left. The second line is a comment '# ...and you are done!'. The third line is empty. The line numbers 1, 2, and 3 are visible on the left side of the editor.

EVALUATION

- Benchmark user scripts from <http://speed.pypy.org>
- Control group: 20 executions of the user script, repeated 20 times
- Treatment group: 100 executions of the user script (about 80 for optimisation, 20 for post-hoc evaluation), repeated 20 times
- Fitness: execution time measured with system clock

Table 1. Benchmark user scripts used for the JIT optimisation case study

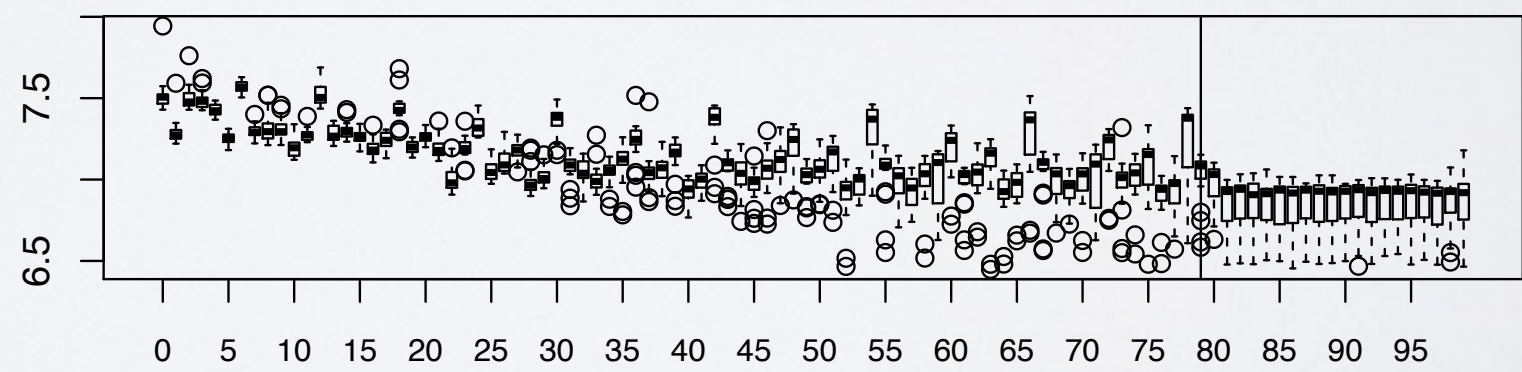
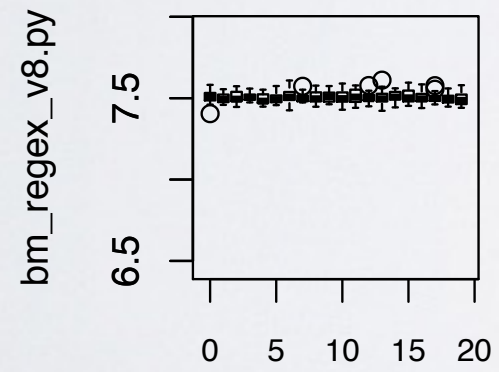
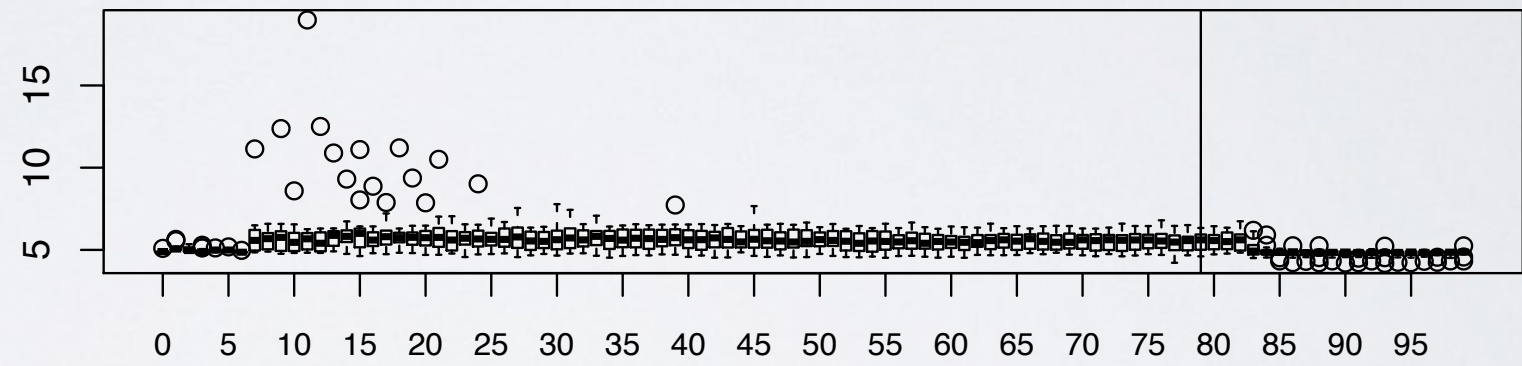
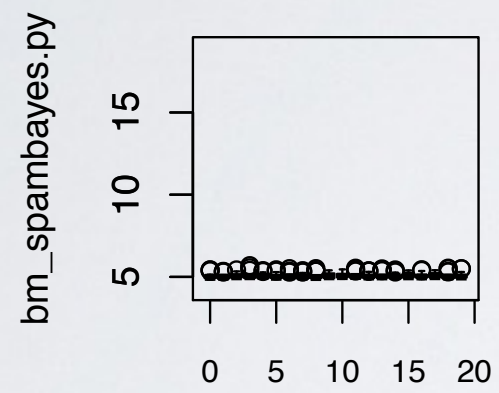
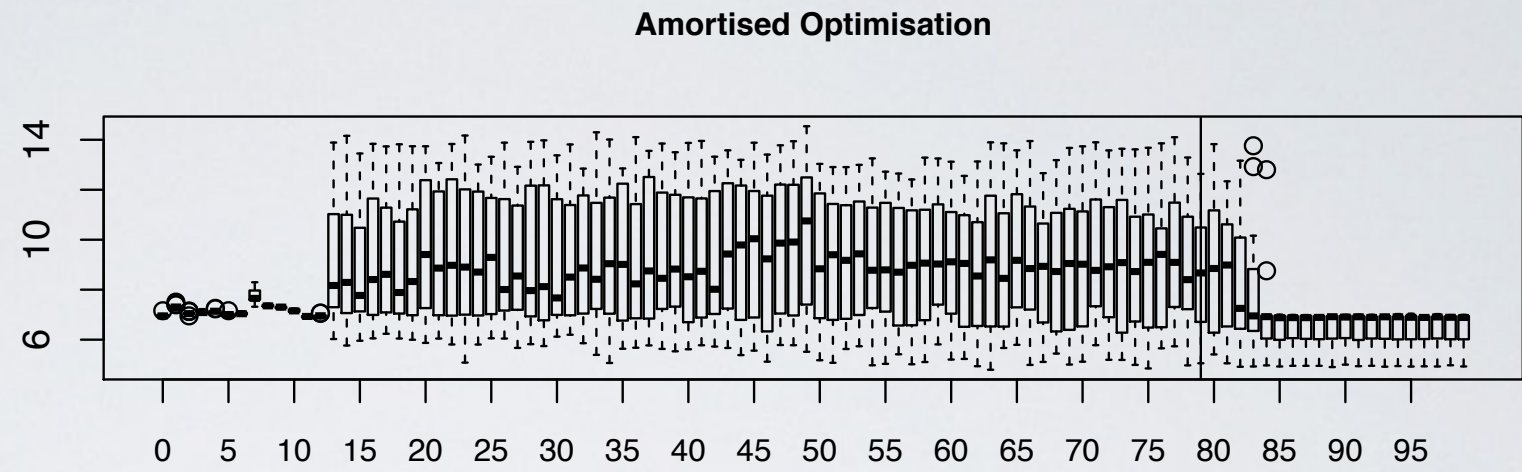
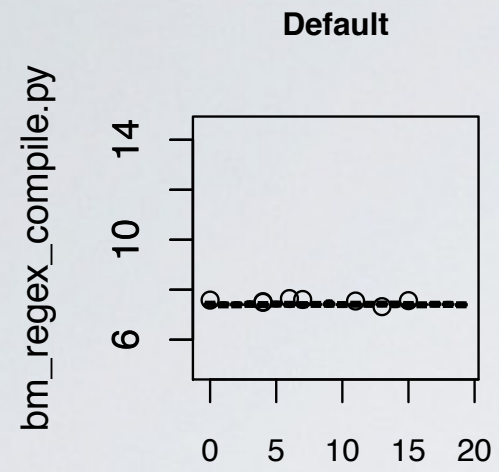
Script	Description
<code>bm_call_method.py</code>	Repeated method calls in Python
<code>bm_django.py</code>	Use <code>django</code> to generate 100 by 100 tables
<code>bm_nbody.py</code>	Predict n -body planetary movements ^a
<code>bm_nqueens.py</code>	Solve the 8 queens problem
<code>bm_regex_compile.py</code>	Forced recompilations of regular expressions
<code>bm_regex_v8.py</code>	Regular expression matching benchmark adopted from V8 ^b
<code>bm_spambayes.py</code>	Apply a Bayesian spam filter ^c to a stored mailbox
<code>bm_spitfire.py</code>	Generate HTML tables using <code>spitfire</code> ^d library

^a Adopted from <http://shootout.alioth.debian.org/u64q/benchmark.php?test=nbody&lang=python&id=4>.

^b Google's Javascript Runtime: <https://code.google.com/p/v8/>.

^c <http://spambayes.sourceforge.net>

^d A template compiler library: <https://code.google.com/p/spitfire/>



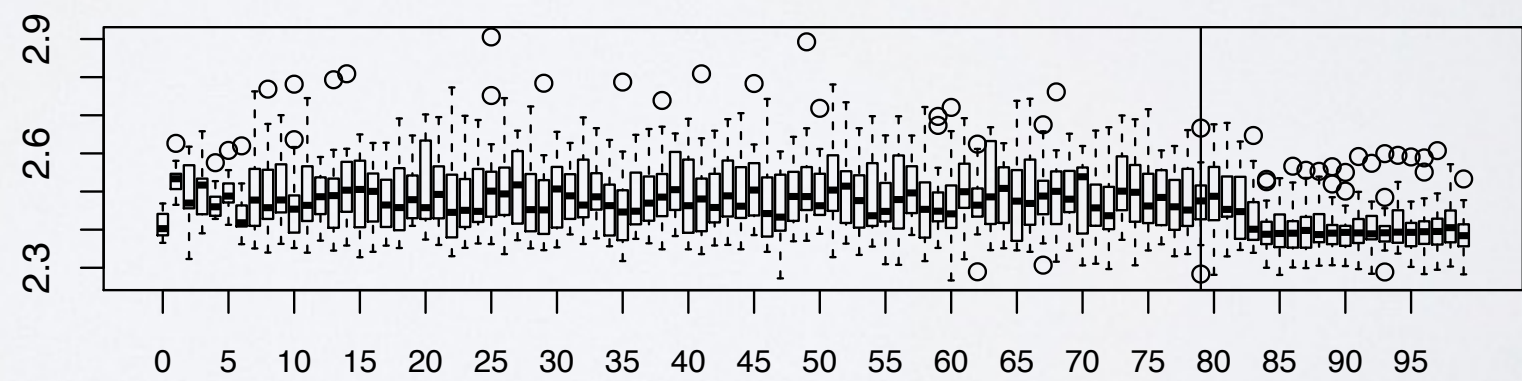
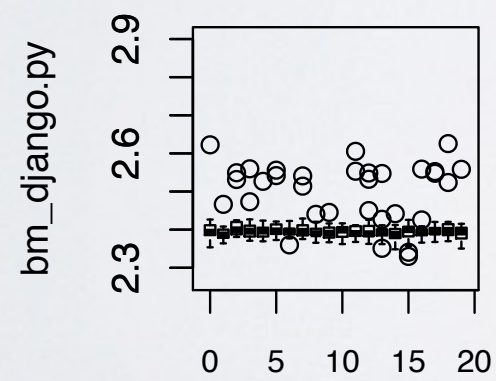
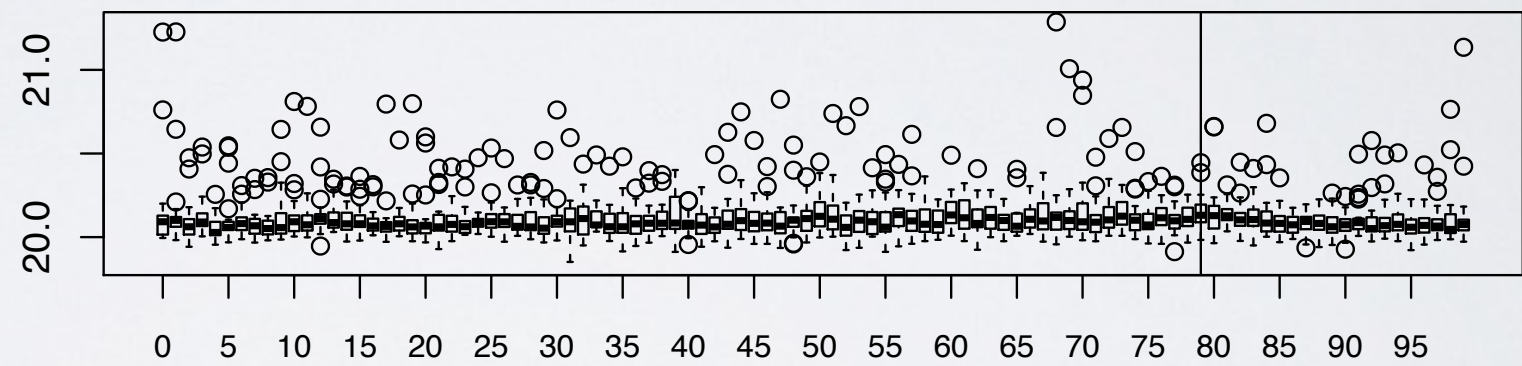
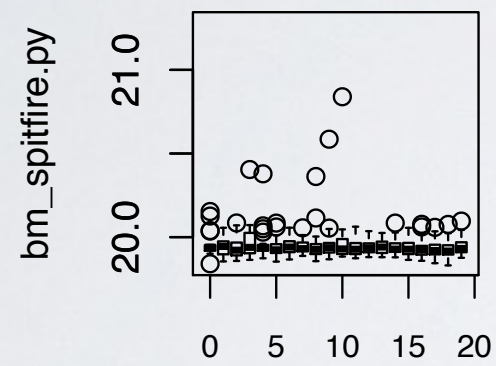
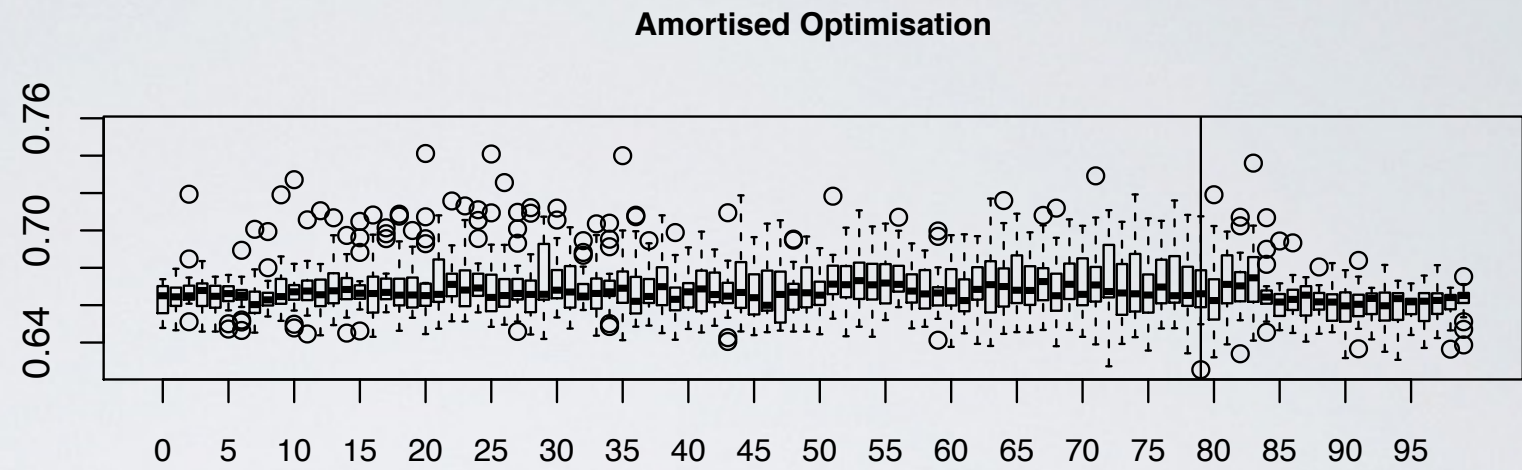
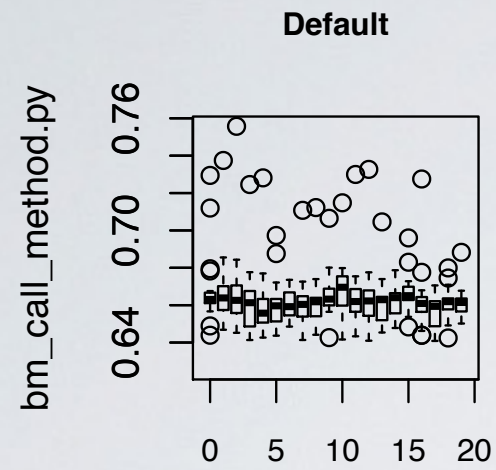


Table 2. Descriptive statistics and the p -values of the hypothesis testing from the pypy case study. The user script `bm_regex_v8.py` becomes 8.6% faster after the amortised optimisation.

Subject	Default		Optimised		p -value
	Mean	Std. Dev.	Mean	Std. Dev.	
<code>bm_call_method.py</code>	0.6631	0.0150	0.6630	0.0130	0.4478
<code>bm_django.py</code>	2.4018	0.0397	2.4161	0.0753	0.9996
<code>bm_nbody.py</code>	1.1948	0.0071	1.1871	0.0136	<1e-4
<code>bm_nqueens.py</code>	2.7367	0.0237	2.5595	0.0743	<1e-4
<code>bm_regex_v8.py</code>	7.5045	0.0347	6.8580	0.1583	<1e-4
<code>bm_regex_compile.py</code>	7.4155	0.0471	6.8786	1.5073	<1e-4
<code>bm_spambayes.py</code>	5.0654	0.1654	4.9346	0.3851	<1e-4
<code>bm_spitfire.py</code>	19.9485	0.0861	20.1045	0.1228	1.0000

HOW ABOUT HARDWARE?

Let us consider matrix multiplication.

$$\left\{ \begin{array}{c} \blacksquare \end{array} \right\} \times \left\{ \begin{array}{c} \blacksquare \end{array} \right\} = \left\{ \begin{array}{c} \blacksquare \end{array} \right\}$$

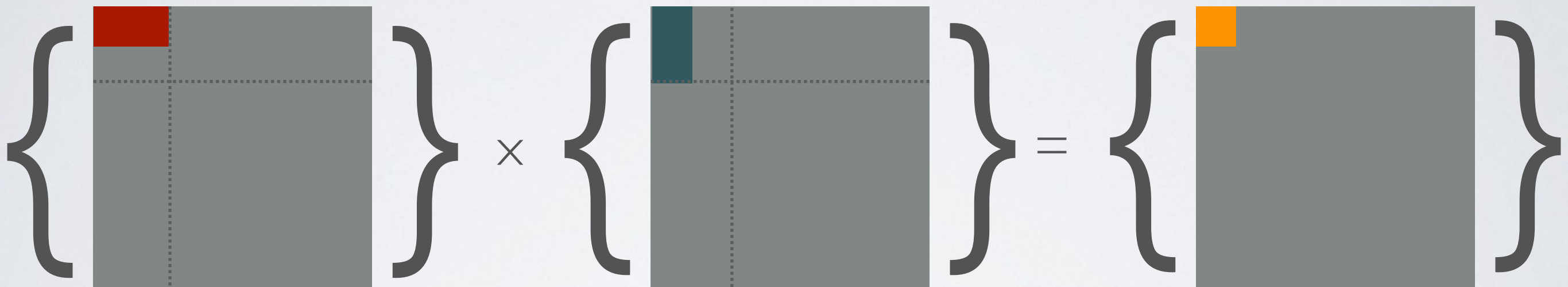
HOW ABOUT HARDWARE?

Let us consider matrix multiplication.



HOW ABOUT HARDWARE?

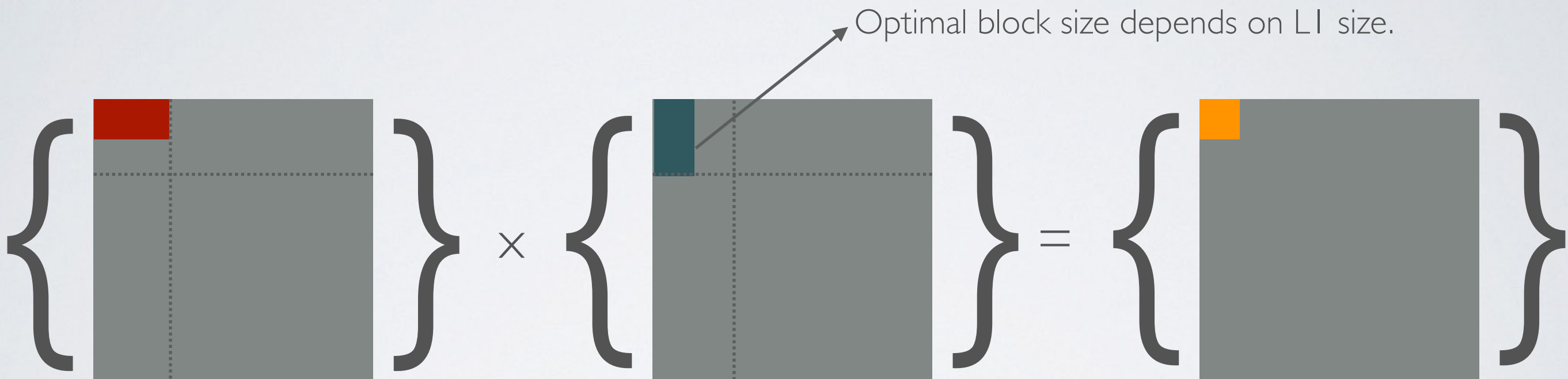
Let us consider matrix multiplication.



Blocked Matrix Multiplication: smaller inner loop
to fit everything into L1 cache.

HOW ABOUT HARDWARE?

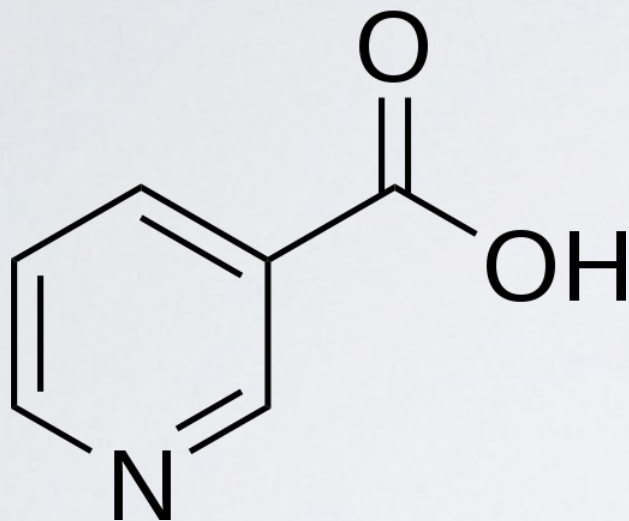
Let us consider matrix multiplication.



Blocked Matrix Multiplication: smaller inner loop
to fit everything into L1 cache.

NIA³CIN

Non-Invasive, Amortised and Autonomous Code Injection



```
BlockedMatrixMultiplication.java
```

```
11  
12 @Input(name = "block_size", initvalue = "2", bound = "1, 512")  
13 public void setBlockSize(int size)  
14 {  
15     this.BLOCK_SIZE = size;  
16 }  
17  
18 @Optimize(name = "rate", type = Double.class, direction = Optimize.Direction.MAXIMIZE)  
19 public Double getRate()  
20 {  
21     return new Double(rate);  
22 }  
23
```

Annotation-based

```
23  
24 public BlockedMatrixMultiplication()  
25 {  
26     Niacin.initialise(BlockedMatrixMultiplication.class);  
27 }  
28
```

Event-driven dependency injection

EVALUATION

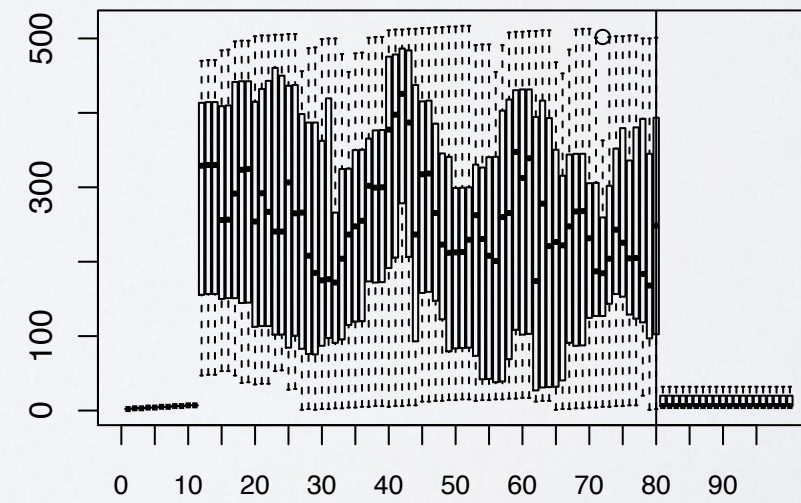
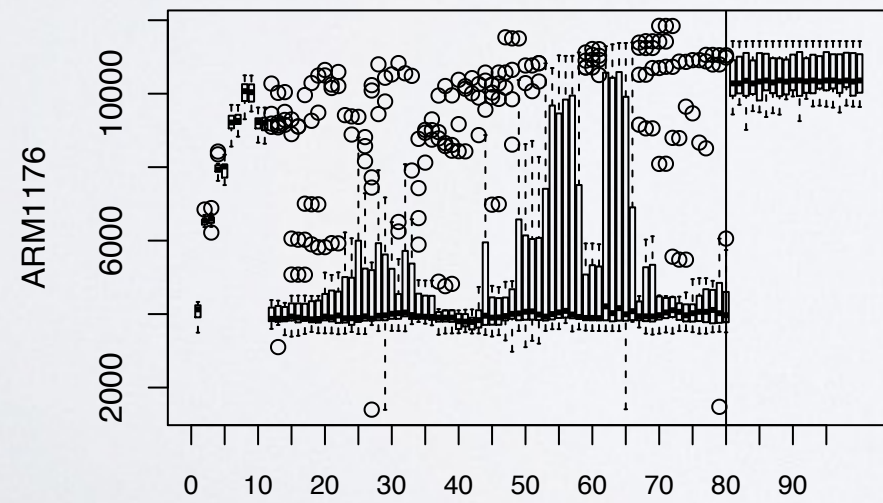
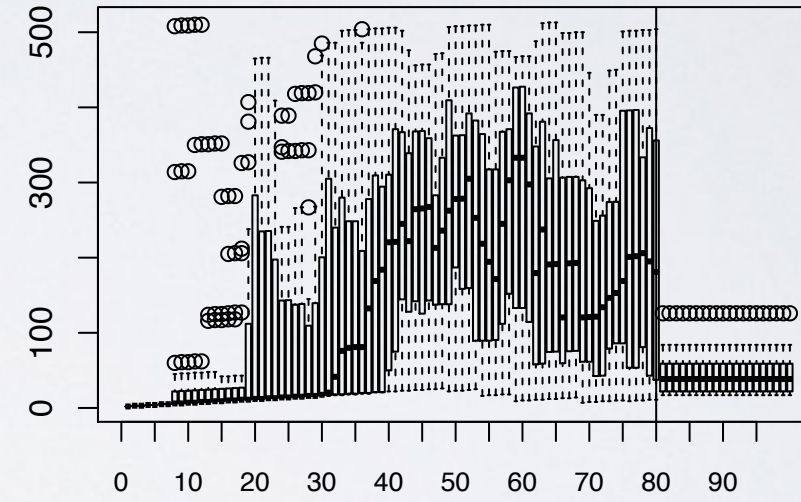
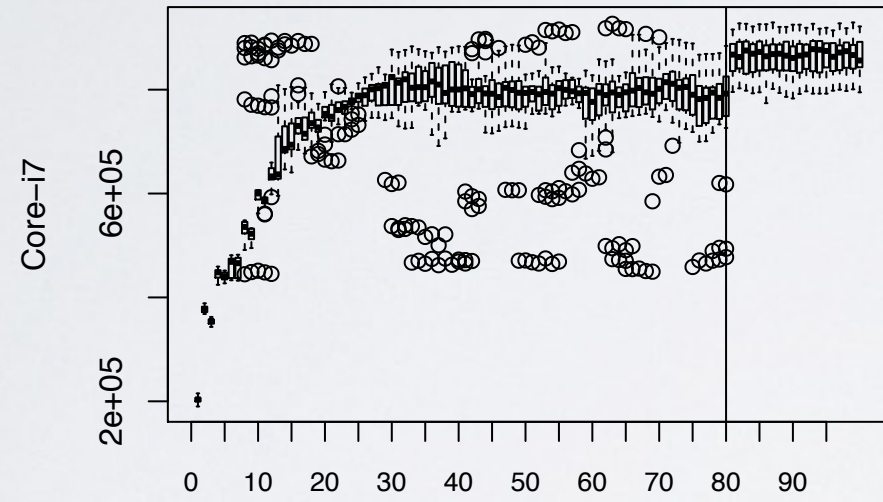
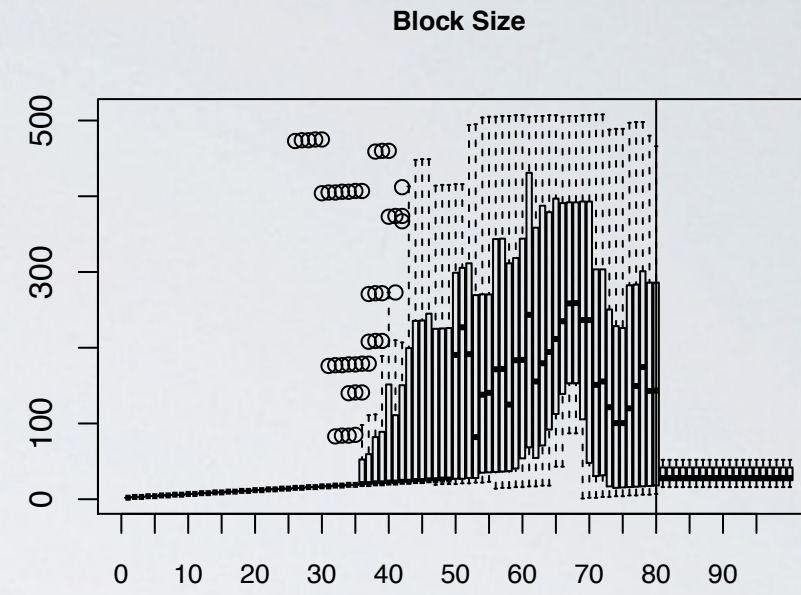
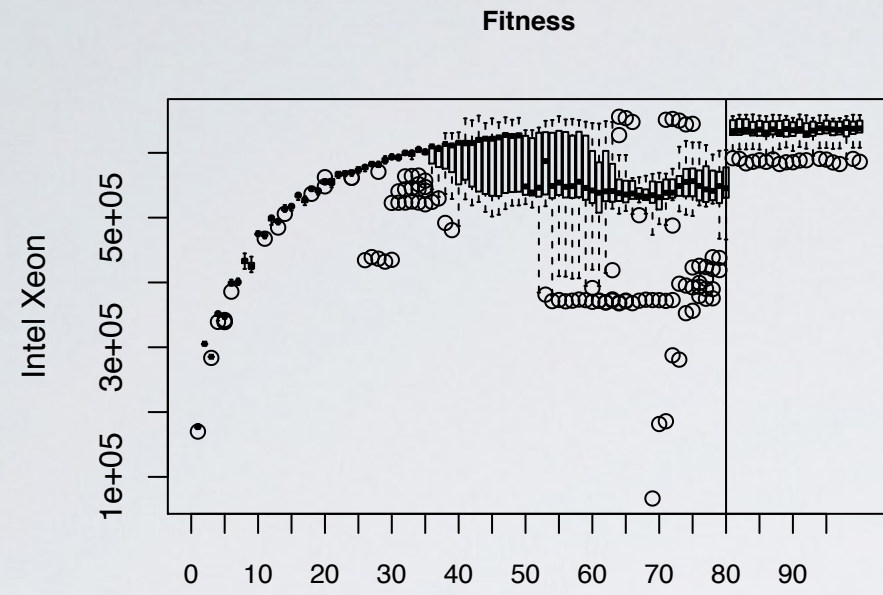


Table 3. Information about CPUs for which BMM was optimised

CPU	Clock Frequency	L1 Instruction Cache	L1 Data Cache
Intel Xeon W3680 ^a	3.33GHz	32KB	32KB
Intel Core-i7 3820QM ^a	2.7GHz	32KB	32KB
ARM1176 (BCM2835 SoC) ^b	250MHz	16KB	16KB

^a These Intel CPUs share data and instruction caches between two processor threads.

^b Raspberry Pi Model B, first edition.



RESULTS

Table 4. Descriptive statistics of the BMM algorithm

CPU	Block Size = 2		Optimised			
	Mean	Fitness Std. Dev.	Mean	Fitness Std. Dev.	Mean Block Size	Std. Dev.
Xeon	305189.00	1118.35	634510.13	17254.99	32.25	10.52
Core-i7	377196.74	6360.66	863878.91	34566.63	44.05	26.85
ARM1176	6531.64	124.07	10486.23	574.29	12.90	8.97

- Core-i7 can use the largest block size. Its single core performance is actually higher than Xeon (confirmed by a 3rd party benchmark).
- ARM can only use the mean block size of roughly 13.

THREATS

THREATS



Restricted to
behaviour-
preserving
optimisations only

THREATS



Restricted to
behaviour-
preserving
optimisations only



Getting precise
measurements

THREATS



Restricted to
behaviour-
preserving
optimisations only



User may experience
performance fluctuation



Getting precise
measurements

THREATS



User may experience
performance fluctuation



Getting precise
measurements



Restricted to
behaviour-
preserving
optimisations only



We want you!

AMORTISED OPTIMISATION



Optimisation executed in micro-steps,
each in-situ execution as a single fitness evaluation

AMORTISED OPTIMISATION



Optimisation executed in micro-steps,
each in-situ execution as a single fitness evaluation

AMORTISED OPTIMISATION



Budget Controlled
(will stop when run out)



Low Overhead
(only microscopes)

AMORTISED OPTIMISATION



Optimisation executed in micro-steps,
each in-situ execution as a single fitness evaluation

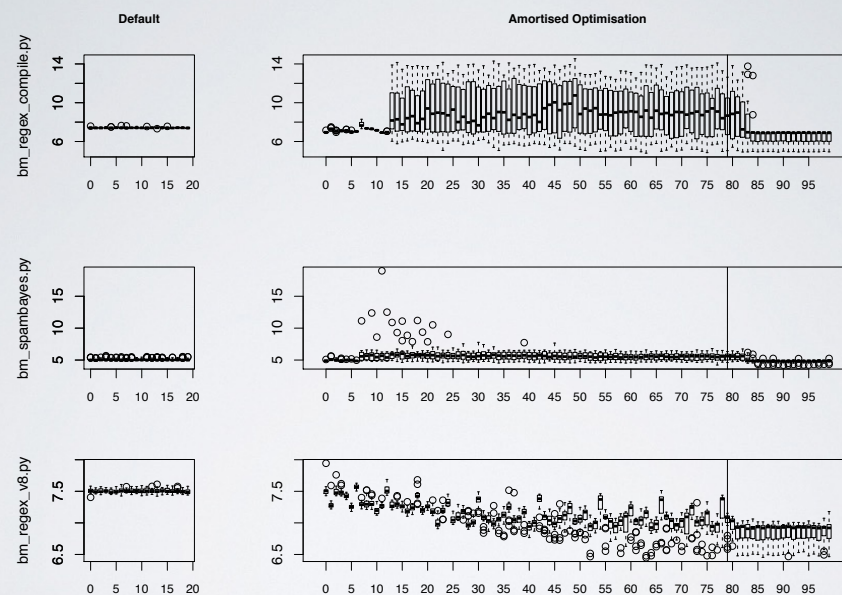
AMORTISED OPTIMISATION



Budget Controlled
(will stop when run out)



Low Overhead
(only microscopes)



AMORTISED OPTIMISATION



Optimisation executed in micro-steps,
each in-situ execution as a single fitness evaluation

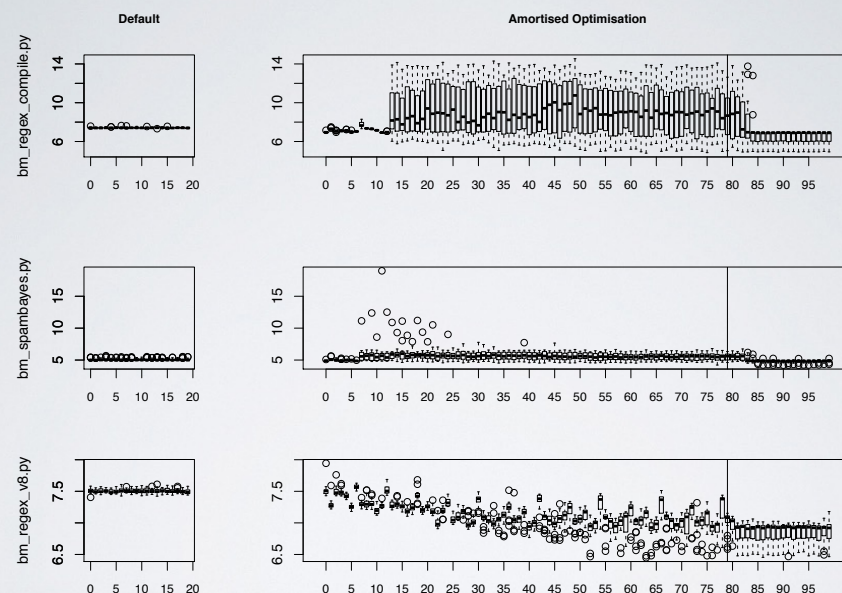
AMORTISED OPTIMISATION



Budget Controlled
(will stop when run out)



Low Overhead
(only microscopes)



Code Available

<https://bitbucket.org/ntrolls/piacin>

<https://bitbucket.org/ntrolls/niacin>