Amortised Optimisation of Non-functional Properties in Production Environments

Shin Yoo(✉)

Korea Advanced Institute of Science and Technology,
291 Daehak-ro, Yuseong-gu, Daejeon 305-338, Republic of Korea
shin.yoo@cs.kaist.ac.kr

Abstract. Search Based Software Engineering has high potential for optimising non-functional properties such as execution time or power consumption. However, many non-functional properties are dependent not only on the software system under consideration but also the environment that surrounds the system. This necessitates a support for online, in situ optimisation. This paper introduces the novel concept of amortised optimisation which allows such online optimisation. The paper also presents two case studies: one that seeks to optimise JIT compilation, and another to optimise a hardware dependent algorithm. The results show that, by using the open source libraries we provide, developers can improve the speed of their Python script by up to 8.6 % with virtually no extra effort, and adapt a hardware dependent algorithm automatically for unseen CPUs.

1 Introduction

Non-functional properties have increasingly been the focus of Search Based Software Engineering (SBSE) work [2]. The inherent dynamic nature of SBSE, i.e. measuring the fitness from actual executions of the subject of optimisation, makes it a powerful tool to deal with non-functional properties. Testing of temporal behaviours have received a considerable amount of interest [4,8,16,18]; other properties like Quality of Service [6,14] and security [5,9,10] are emerging fields of research.

Most existing literature on non-functional properties concerns what can be called *offline* optimisation: we define an optimisation problem to improve a specific non-functional property, and consequently obtain one or more solutions by using meta-heuristic optimisation algorithms, which are then deployed. This approach overlooks an important and challenging element of non-functional properties: environmental dependency. Non-functional behaviours of software systems are hard to predict precisely because they are heavily affected by the various environmental factors ranging from operational profiles of input data to the hardware that runs the system. By performing the optimisation offline, we detach the subjects from their environments and tailor our solution to the specific environment in which we optimise.

M. Barros and Y. Labiche (Eds.): SSBSE 2015, LNCS 9275, pp. 31–46, 2015.
DOI: 10.1007/978-3-319-22183-0_3

This offline approach raises two issues about the quality of the resulting solutions. First, it is difficult to avoid sampling bias. Recreating the production environment precisely can be difficult, because certain factors are either highly variable (e.g. hardware components), or hard to emulate (e.g. realistic user load for web applications). Consequently, offline optimisation can introduce bias that favours the often limited optimisation environment. Second, even when the offline optimisation is satisfactory, the production environment may change in such a way that degrades the behaviour of the deployed system (e.g. upgraded hardware with different performance profiles). This necessitates that the system is taken offline and re-optimised, which may be difficult in industrial settings.

One way to overcome these problems is to provide built-in adaptivity in the deployed software, so that the optimisation can take place in the production environment after deployment. Since we will be optimising in the real environment, there cannot be any sampling bias. Because the adaptivity is built-in, there is no need to take the system offline to optimise for the changed environment; the system will continue to adapt to changes. Naturally, the focus is on how to perform the optimisation without damaging the performance of the system in the production environment.

We introduce a novel concept called amortised optimisation. Executions of any metaheuristic optimisation can be amortised across multiple fitness evaluations. Normally, optimisation algorithms perform fitness evaluations either one by one (if it is a local search) or as a group (a population-based algorithm). With amortised optimisation, it is the fitness evaluation that drives the optimisation algorithm. Whenever the System Under Metaheuristic Optimisation (SUMO) is executed, we measure one fitness value out of it, and drive the optimisation forward by a single step. One iteration of the optimisation – either the evaluation of neighbours and the move to a better neighbour (a local search), or the evaluation of an entire population and the move to the next generation (a population-based algorithm) – will consist of multiple executions of SUMO.

The paper investigates this novel approach to optimisation of non-functional properties through two case studies. The first concerns adapting to different software: we apply the amortised optimisation to improve Just-In-Time (JIT) compilation parameters in a state-of-the-art Python runtime, `pypy` [3], and measure the impact on speed using benchmarks. For the `pypy` runtime, this can be seen as adapting to Python scripts it has not executed before. The second study focuses on hardware differences: we apply amortised optimisation to improve blocked matrix multiplication [7], whose performance depends on the combination of block size parameter and the size of the L1 cache in the CPU that executes the blocked algorithm. From the point of the algorithm, this can be seen as adapting to a CPU that it has not been executed on before. Both studies are supported by open source implementations of amortised optimisation techniques. The results show that amortised optimisation can improve non-functional properties of SUMO without knowing the details of the production environment in advance.

The contributions of this paper are as follows:

- **Amortised Optimisation:** we introduce the concept of amortised optimisation, which takes place over multiple executions of SUMO in order to reduce the optimisation overhead with each execution. We make open source libraries for Java and `pypy` JIT optimisation available.
- **Empirical Evaluation:** we present two exploratory case studies of the application of amortised optimisation, focusing on software and hardware differences respectively. The first study seeks to optimise JIT compilation parameters of a Python runtime, without knowing which scripts will be executed in advance. The second study aims to optimise the block size in blocked matrix multiplication, without knowing which CPU the algorithm will be executed on in advance.

The rest of the paper is organised as follows. Section 2 introduces the concept of amortised optimisation. Section 3 presents the case study on JIT compilation parameters, while Sect. 4 presents the case study on blocked matrix multiplication. Section 5 discusses the related work, and Sect. 6 concludes.

2 Amortised Optimisation

Many of non-functional properties of software depend on the exact context and environment it is being used in. Consequently, the best way to adapt to different contexts and environments is to optimise these properties *in situ*. However, metaheuristic optimisation often relies on a non-trivial number of fitness evaluations, which, in the context of Search-Based Software Engineering, may contain other software, model, or even hardware in the loop [12]. The inhibitive cost effectively prevents software to be optimised in the production environment.

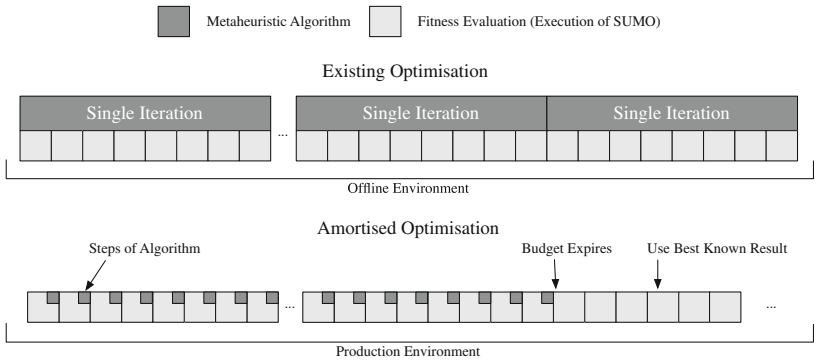

Fig. 1. Amortised optimisation interleaves executions of SUMO, from which the fitness is measured, with partial executions of metaheuristic algorithms. Each normal use of SUMO doubles as a single fitness evaluation, driving the optimisation by small steps. When the initial budget for optimisation expires, SUMO simply continues to run with the best known result.

We posit that the cost of in situ optimisation can be amortised. Figure 1 presents the conceptual overview of the amortised optimisation approach. With existing optimisation techniques (depicted at the top), algorithms perform multiple iterations, each of which, in turn, executes the SUMO to measure the fitness. This process is performed in the offline environment, and the result is deployed. With amortised optimisation (depicted at the bottom), a single iteration of a metaheuristic algorithm is broken down into several smaller steps to be executed as part of each execution of SUMO. Essentially, we seek to pause and resume the metaheuristic algorithms with persistence support. By keeping each step smaller, we minimise the computational overhead to the normal operation of SUMO. Gradually, SUMO will find better solutions: when the budget for optimisation runs out, SUMO can continue to use the best known result.

Any metaheuristic algorithm can be amortised in the proposed way, because there is little dependency between the algorithm itself and the execution of SUMO for fitness evaluation. A more limiting factor would be the nature of the optimisation problem: since we are to explore the search space with the actual uses of SUMO, we cannot afford to functionally sabotage any execution. For example, a suboptimal candidate solution may be allowed to slow the software a little bit, or use more memory than usual. However, it cannot affect the functionality of SUMO so that it produces incorrect output. Consequently, amortised optimisation is more easily applicable to tuning performance-related parameters than to perform Genetic Improvement that may crash the SUMO [13]. For the latter, parallel execution of two instances of SUMO may provide a solution: such parallel execution has been previously studied to recover from regression faults while the system is running online [11].

2.1 State-Based Steps: A Hill Climbing Example

Let us present a high-level model of amortised hill climbing algorithm, which is shown in Fig. 2 in a state-based model format. Vertices represent the state the algorithm can be in; edges represent potential control flows between algorithm states. Edge labels are written in the format of `X/Y`, where `X` denotes a transition trigger and `Y` denotes the set of actions performed during the transition. Variable `eval` keeps track of the number of remaining fitness evaluations available to the algorithm. N is a set of neighbouring solutions: Next(N) iterates over neighbours, while HasNext(N) checks whether the iteration is over. IsLocalOptima() checks whether the current candidate solution is a local optimum. Finally, return(x) returns x as a candidate solution for the SUMO to use. Note that we assume a feedback loop from the SUMO back to x, which provides the fitness value: this is not depicted in Fig. 2.

Whenever the SUMO is executed, it asks for a candidate solution x from the amortised optimisation. The amortised hill climbing algorithm first retrieves its current status from the persistence layer, then executes transitions until it makes a return(x) call. For example, when the SUMO with the amortised hill climbing algorithm is executed for the first time, it will start in the initial node ("Random Solution"): since $eval > 0$ at the beginning, a transition is triggered,

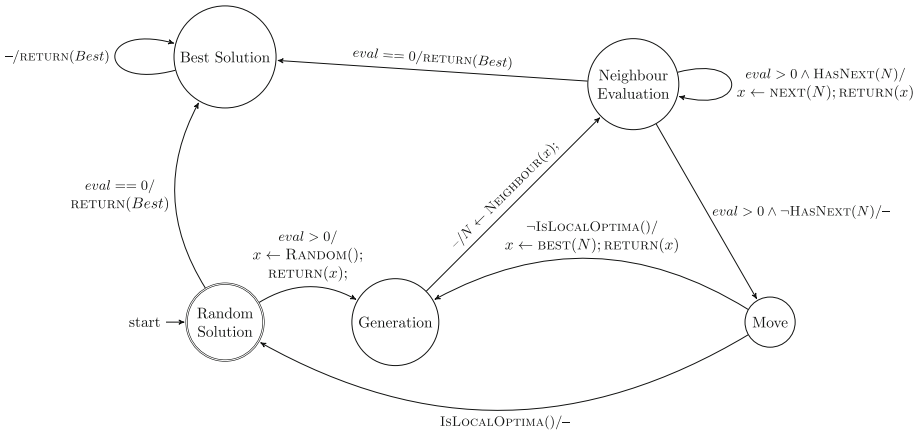

Fig. 2. State-based model of amortised hill climbing algorithm: $return(x)$ decreases the remaining number of fitness evaluations, $eval$, by 1.

and the algorithm returns the randomly generated x. Finally, it records the current state ("Generate") to the persistence layer and pauses. The next time the SUMO is executed, the algorithm resumes itself from the stored status, and the next transition (to "Evaluate Neighbours") is immediately triggered, while generating neighbours of the current x. This time, the algorithm only pauses when the second transition is triggered and the call RETURN(x) is made (which returns the first solution in N).

3 Case Study: Optimising the JIT Parameters for `Pypy`

3.1 JIT Parameter Optimisation

`Pypy` is a Python runtime implementation with a strong focus on Just In Time (JIT) compilation [3]. The JIT compilation mechanism used by `pypy` is the meta-tracing JIT. Tracing JIT starts by profiling the code to identify frequently executed, or *hot*, loops. In the next stage, the runtime records the history of all operations executed during a single iteration of a hot loop. These are then translated into the native machine code. What is unique with `pypy` is that the tracing JIT is not applied to the user script, but rather to the interpreter that runs the user script (hence the name, *meta*-tracing).

How aggressively `pypy` tries to JIT compile the user script depends on a set of parameters that control the behaviour of the tracing JIT. While JIT compilation in general can make Python, which is interpreted, significantly faster with `pypy`, it is not always the case that JIT compiling more of the user script results in shorter execution time. The more aggressive `pypy` tries to JIT compile, the higher the cost of tracing becomes. If the gain in JIT compilation does not exceed the cost, compiling more of the user script can actually slow `pypy` down. This trade off is unique to each user script and the environment `pypy` runs in. Therefore,

finding the desirable set of JIT parameters for `pypy` can be an ideal application for the in situ, amortised optimisation.

Among the parameters that control the behaviour of the `pypy` JIT, the case study in this paper focuses on the following three:

- **Function threshold:** This parameter determines the number of times a function has to be executed before it is traced from the beginning. Default value in `pypy` is 1619.
- **Loop threshold:** This parameter determines the number of times a loop has to be executed before it is identified as a hot loop. Default value in `pypy` is 1039.
- **Trace eagerness:** To ensure correctness, `pypy` inserts *guards* in the translated code. When guards fail (an unpredicted branching direction can be a potential cause), tracing JIT falls back to interpreting the loop. If guard failure happens above certain threshold, tracing JIT attempts to translate the sub-path from the point of guard failure to the end of the loop (this is called a *bridge*). This parameter determines the number of times a guard has to fail before `pypy` compiles the bridge. Default value in `pypy` is 200.

These parameters have been chosen after consulting the developers of `pypy`. We have also been advised to set the loop threshold to be smaller than the function threshold. Consequently, the implementation of amortised optimisation of JIT for `pypy` replaces the loop threshold parameter with a threshold ratio parameter, whose value is within $(0, 1)$. The actual loop threshold parameter is set to [function threshold] $\cdot$ [threshold ratio]. For function threshold, we use the range of $[10, 4900]$; for trace eagerness, we use the range of $[1, 1000]$.

3.2 Experimental Setup

Benchmarks. We chose 8 benchmark scripts from the standard benchmark suite with which the speed of `pypy` is evaluated [17]. Table 1 describes the user script studied in this paper.

Each benchmark script contains a main test function that performs the operation described in Table 1. The scripts have been slightly modified to repeat their main test functions 50 times with each execution: this is to overcome the inherent randomness in measuring execution times. The execution time is measured using the system clock, starting from the invocation of the test function, and ending when it returns. It does not include any time used by the amortised optimisation itself. The rationale is twofold: the overhead for a single execution of the user script is very light, and when the amortised optimisation finishes (i.e. runs out of the allocated fitness evaluations), it becomes virtually zero.

Implementation. The amortised optimisation for JIT parameters is implemented into a Python package called `piacin`[1]. Since the JIT parameters only

[1] `Piacin` is made available as open source software at https://bitbucket.org/ntrolls/piacin.

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

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

need to be set once during the execution of a single user script, `piacin` similarly only needs to be called twice: when the user script starts (to configure `pypy` with the current parameters), and when it finishes (to record the fitness value associate with the current parameters). The first hook is implemented by implementing `piacin` as a Python package, and placing the JIT configuration code as part of the package initialisation. The second hook is implemented by using the `atexit` hook provided by Python by default. The benefits of this package-based design is that the user only needs to include `piacin` package (i.e. to have `import piacin` at the beginning of the user script) to benefit from it.

The amortised optimisation algorithm in `piacin` is steepest ascent hill climbing. Neighbourhood solutions are generated by adding and subtracting predefined step values to each of the parameters: 20 for function threshold, 10 for trace eagerness, and 0.05 for threshold ratio. When the newly generated candidate solution has any parameter outside the predefined range, the parameter value is wrapped around the range.

We use the default parameters of `pypy` as the starting point of the hill climbing. Since these parameters are the result of careful benchmarking, it would be wasteful to discard them without consideration. However, when the hill climbing reaches local optima, we fall back to the random restart mechanism.

Control vs. Treatment Group. The control group consists of 20 un-optimised runs of user benchmark scripts. Each control group run contains 20 un-optimised `pypy` executions of the corresponding scripts. The treatment group consists of 20 optimised runs of user benchmark scripts. Each treatment group run contains 100 optimised `pypy` executions of the corresponding scripts: 80 executions at the beginning are used for optimisation, the best solution from which is used by the remaining 20 executions. Both groups have been executed with `pypy` version

2.4.0 on Mac OS X 10.10.2, using Intel Xeon 3.3 Hz CPU with 6 cores and 16 GB of RAM. All the user scripts are single threaded and were executed one by one.

3.3 Results

Figures 3 and 4 show the boxplots of 20 runs of both control and treatment groups. The x-axis shows the sequence of repeated executions of user scripts in each run; the y-axis shows the execution time in seconds. Visual observation reveals that, for some user scripts, the execution time after the amortised optimisation can be indeed shorter than before: optimisation for `bm_regex_v8.py` shows a very clear trajectory with improving fitness (i.e. decreasing execution time), while `bm_nbody.py`, `bm_nqueens.py`, `bm_regex_compile.py`, and `bm_spambayes.py` settle down with shorter execution times after exploring the search space during the optimisation.

With some user scripts, such as `bm_nbody.py` and `bm_nqueens.py`, the very first execution of the user script during amortised optimisation already shows shorter execution time. This appears to be counter-intuitive, as the parameters are the same as the default ones when the amortised optimisation runs begin.

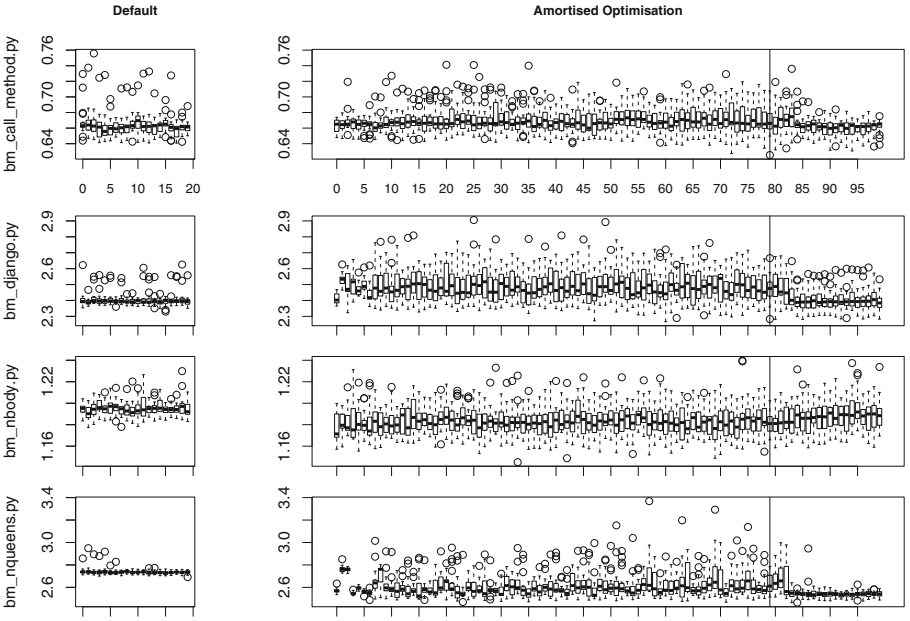

Fig. 3. Boxplots of execution times of user scripts with and without amortised optimisation applied to `pypy`. Plots on the left shows the execution times of benchmark scripts from 20 separate runs, each of which repeats the script 20 times. Plots on the right shows 20 runs, each of which repeats the script 100 times. The first 80 executions are used for the amortised optimisation; the remaining 20 executions show the results of the optimised JIT parameters.

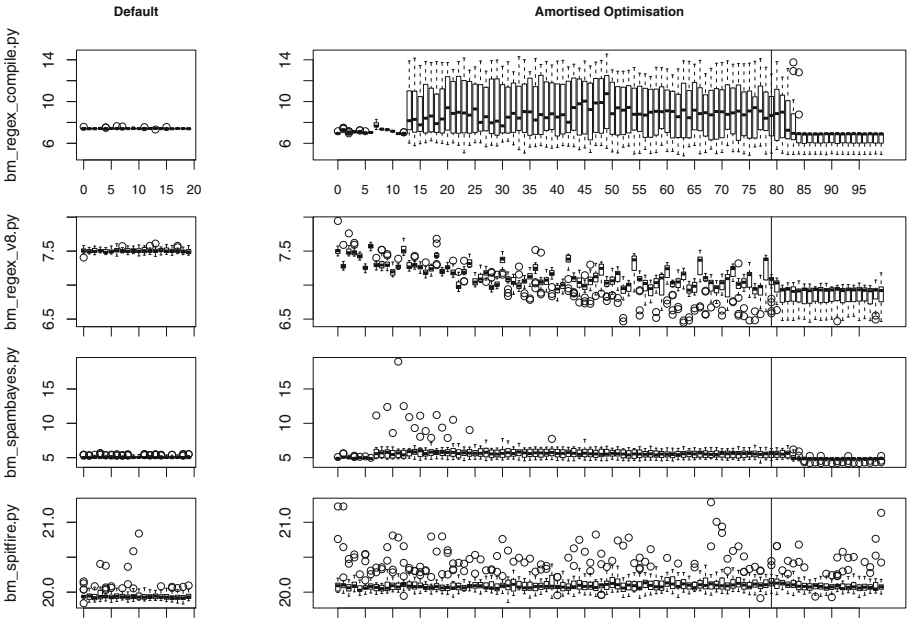

Fig. 4. Boxplots of execution times of user scripts with and without amortised optimisation applied to `pypy`. Plots on the left shows the execution times of benchmark scripts from 20 separate runs, each of which repeats the script 20 times. Plots on the right shows 20 runs, each of which repeats the script 100 times. The first 80 executions are used for the amortised optimisation; the remaining 20 executions show the results of the optimised JIT parameters.

This is explained by the fact that, when `piacin` is applied, `pypy` does execute a little bit more Python code (that belongs to `piacin`) before the benchmark test function is invoked. Since the tracing JIT in `pypy` is applied to the interpreter of Python rather than the user script, this extra Python code will inevitably make certain parts of Python interpreter within `pypy` *hotter* than when the user script is executed without `piacin`.

Table 2 contains both the descriptive statistics and the results of the hypothesis testing. Execution times of both the default and optimised runs passed the Shapiro-Wilk normality test; consequently, we use mean and standard deviation as descriptive statistics and t-test to test the alternative hypothesis that the execution times from the optimised runs are shorter than those from the default ($\alpha = 0.05$). The results of the hypothesis tests confirm the visual observation, as five user scripts show shorter execution time that are statistically significant. In case of `bm_regex_v8.py`, the optimised runs are faster by 8.6 %.

Table 2. Descriptive statistics of the execution time (in seconds) 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.	
`bm_call_method.py`	0.6631	0.0150	0.6630	0.0130	0.4478
`bm_django.py`	2.4018	0.0397	2.4161	0.0753	0.9996
`bm_nbody.py`	1.1948	0.0071	1.1871	0.0136	**<1e-4**
`bm_nqueens.py`	2.7367	0.0237	2.5595	0.0743	**<1e-4**
`bm_regex_v8.py`	7.5045	0.0347	6.8580	0.1583	**<1e-4**
`bm_regex_compile.py`	7.4155	0.0471	6.8786	1.5073	**<1e-4**
`bm_spambayes.py`	5.0654	0.1654	4.9346	0.3851	**<1e-4**
`bm_spitfire.py`	19.9485	0.0861	20.1045	0.1228	1.0000

4 Case Study: Optimising Algorithms to Hardware

The second case study concerns the optimisation of performance critical parameter against different hardware components. The subject algorithm is the Blocked Matrix Multiplication (BMM).

Algorithm 1. BMM
Input: Size of matrices, n, n-by-n matrices
A and B
Output: matrix C, which equals to $A \cdot B$
(1) $n\_blocks \leftarrow \lceil \frac{n}{BS} \rceil$
(2) **for** $b_i = 0$ **to** $b_i < n\_blocks$
(3) $i \leftarrow b_i * BS$
(4) **for** $b_j = 0$ **to** $b_j < n\_blocks$
(5) $j \leftarrow b_j * BS$
(6) **for** $b_k = 0$ **to** $b_k < n\_blocks$
(7) $k \leftarrow b_k * BS$
(8) BLOCK(n, A, B, C, i, j, k)

Algorithm 2. BLOCK
Input: Matrix size, n, matrices A, B, and C,
indices i, j, and k
Output: Updates matrix C
(1) $\_M \leftarrow (i + BS > n?n - i : BS)$
(2) $\_N \leftarrow (j + BS > n?n - j : BS)$
(3) $\_K \leftarrow (k + BS > n?n - k : BS)$
(4) **for** $\_i = 0$ **to** $\_i < \_M$
(5) **for** $\_j = 0$ **to** $\_j < \_N$
(6) $\_cij \leftarrow C[j + i * n + {}_j + {}_i * n]$
(7) **for** $\_k = 0$ **to** $\_k < \_K$
(8) $\_cij += A[i \cdot n + k + \_i \cdot n + \_k] \cdot$
(9) $B[j + k \cdot n + \_j + \_k \cdot n]$
(10) $C[j + i \cdot n + \_j + \_i \cdot n] = \_cij$

4.1 Blocked Matrix Multiplication (BMM)

Algorithms 1 and 2 collectively present the Blocked Matrix Multiplication for square matrices. Algorithm 1 breaks down the matrices into smaller blocks of size `BS` (Block Size), and invokes Algorithm 2 for each of them. The introduction of additional loops may appear harmful to performance. However, having nested loops around a smaller region of memory allows BMM to exploit better CPU pipelining and higher cache hit rate, resulting in faster overall computation.

The key to the increased performance is the size of the block. However, choosing the ideal size depends on details of the hardware environment, such as the size of the L1 cache. Hard-coding a fixed block size into BMM may produce desirable performance on one machine, but if the code is deployed to and executed on another machine with a different CPU, there is no guarantee that the same performance will be retained. This provides a compelling use case for amortised optimisation.

4.2 Experimental Setup

Implementation. We use a Java implementation of the BMM algorithm for matrices of `double` type. The amortised optimisation framework, called NIA^{3}CIN (Non-Invasive Amortised and Automated Adaptivity Code Injection), is based on the hill climbing algorithm and is also implemented in Java[2]. To be as unintrusive as possible, NIA^{3}CIN uses a publish-subscribe style event bus to establish communication between the SUMO and the optimisation. Parameters to be optimised (in the case study, the block size), as well as the measure of the fitness (in the case study, the number of floating point multiplications performed per millisecond), need to be marked with annotation. Before the parameter is to be used, the SUMO needs to call NIA^{3}CIN so that the parameter variable is updated with the current solution; after the parameter has been used, the SUMO needs to call NIA^{3}CIN so that the fitness is fed back to the optimisation.

The range of block size was set to $[1, 512]$. NIA^{3}CIN generates neighbouring solutions by adding and subtracting 1 to the current block size. When moving through consecutive block sizes, certain sizes will be evaluated twice: first as the current solution, and second as a neighbour. Since the non-functional fitness measure is expected to be noisy, the redundant behaviour was left in NIA^{3}CIN deliberately, providing opportunities to evaluate the same solution more than once (and, therefore, getting clearer measures of fitness).

Environment. Table 3 shows three different CPUs for which the BMM algorithm was optimised in this study. Intel Xeon is a 6 code desktop CPU with 32 KB instruction and data cache; the Core-i7 used for this study is a mobile (laptop) version, which has the same cache provision as the Xeon CPU. Finally, to investigate how well the amortised optimisation can adapt to an environment

[2] NIA^{3}CIN is made available as open source software at https://bitbucket.org/ntrolls/niacin.

Table 3. Information about CPUs for which BMM was optimised

CPU	Clock frequency	L1 instruction cache	L1 data cache
Intel Xeon W3680[a]	3.33 GHz	32 KB	32 KB
Intel Core-i7 3820QM[a]	2.7 GHz	32 KB	32 KB
ARM1176 (BCM2835 SoC)[b]	250 MHz	16 KB	16 KB

[a]These Intel CPUs share data and instruction caches between two processor threads.
[b]Raspberry Pi Model B, first edition.

with very limited resources, we use the ARM1176 core on a Broadcom BCM2835 System-on-Chip, which is found in Raspberry Pi version 1 model B. Both Intel CPUs ran OS X 10.10.2 and Java SE Runtime (build 1.8.0_25-b17) with the HotSpot 64-Bit Server VM (build 25.25-b02); Raspberry Pi ran Linux 3.18.8 and Java SE Runtime (build 1.8.0-b132, mixed mode) with the HotSpot Client VM (build 25.0-b70, mixed mode).

Data Collection. For this study, to have a control group without the amortised optimisation would mean to execute the BMM algorithm with a fixed arbitrary block size, which would contribute little to investigating how the optimisation can help. Instead, we fixed the starting block size to 2 and repeated matrix multiplications for 100 times on different CPUs: 80 multiplications have been used by the amortised optimisation to search for the best block size, while the remaining 20 multiplications used the known best block size. This process was repeated for 20 times per CPU to cater for the inherent randomness in the algorithm. On Intel CPUs, we used matrices of size 1,000 by 1,000; on the Raspberry Pi, we used matrices of size 500 by 500. The fitness value is measured by the number of floating point operations per millisecond, using the system clock.

4.3 Results

Figure 5 shows the results of the amortised optimisation of the BMM algorithm for different CPUs. The boxplots on the left show how the fitness value (the number of floating point operations per millisecond) across the 20 different runs (x-axis represents the number of times the BMM is executed). The boxplots on the right show which block size was tried: although the hill climbing algorithm relies on the random restart at different points in different runs, these boxplots still reveal interesting trends in the optimisation of the block size. The vertical lines depict the point at which the optimisation stops and the BMM starts using the best known solution.

Both Xeon and Core-i7 benefit from larger block size, up to around 30, which can be observed from the relatively smooth shapes formed of individual boxplots and the straight, consistent increase in the block size in executions 1 to 30. Block sizes from ARM1176 show a much wider exploration of the search space, which did not necessarily result in increased fitness value. For all three CPUs, both

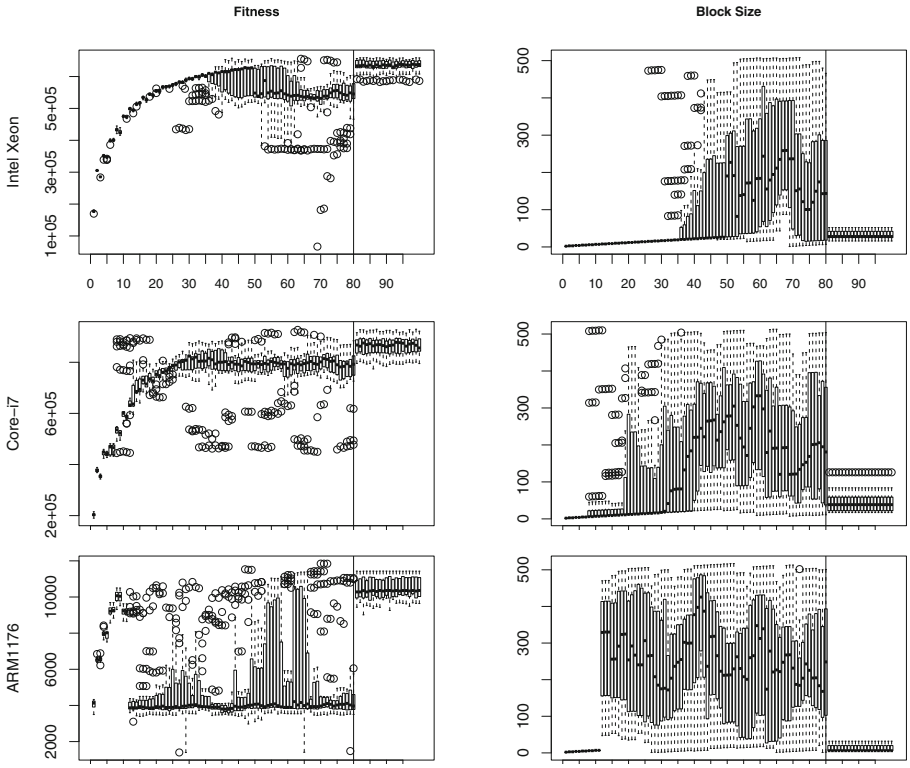

Fig. 5. Changes in the fitness (i.e. number of floating point operations per millisecond) and the block size from 20 runs of BMM for different CPUs. BMM on both Xeon and Core-i7 benefits from increasing block sizes up to around 30. BMM on ARM1176 performs better with much a smaller block size.

the fitness values and the block sizes show relatively small dispersion, suggesting that the optimisations did converge.

Table 4 shows the descriptive statistics of the BMM algorithm, before (i.e. of the first executions of each of the 20 runs) and after the amortised optimisation (i.e. of the last 20 executions of each of the 20 runs). Both the fitness values and the block sizes passed Shapiro-Wilk normality test. For all CPUs, the amortised optimisation can significantly increase the performance of the BMM. An interesting observation is the comparison of Xeon and Core-i7. Despite the higher clock frequency, Xeon performs fewer floating point operations per millisecond. While seemingly counter-intuitive, this shows that NIA^{3}CIN exploits the capabilities of each CPU appropriately. The Xeon W3680 is an older model than the Core-i7 3820QM, and the Core-i7 has been shown to outperform the Xeon in a single core benchmark [1].

Note that this optimisation has been performed automatically and *while* the BMM was operating correctly, using 80 executions. More importantly, if the

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

algorithm is deployed onto a different CPU, the optimisation can start again simply by assigning more budget for fitness evaluations. This can be a significant reduction in effort compared to manual trial and error approach.

5 Related Work

Existing SBSE work that seek to improve non-functional properties almost exclusively uses offline optimisation. Langdon and Harman improved a non-functional property of a non-trivial C++ program using Genetic Programming (GP) [13]. The GP modified several lines in the source code of the original program, making it 70 times faster on average while being as good as the original semantically. The GP-based improvement is much more profound than changing the value of a variable, as it actually patches the source code. However, it also required a significant amount of computation time for off-line optimisation. GP has also been applied to specialise the `MiniSAT` solver for specific problem instances [15]. Wu et al. optimised the behaviour of the dynamic memory allocation in C programs by revealing and optimising hidden parameters [19]. While the aspect of parameter optimisation bears similarity to this paper, Wu et al. also optimised the SUMO in an offline environment. As far as we know, this is the first work that injects the optimisation into the SUMO so that the non-functional properties can be optimised in situ.

6 Conclusion

This paper introduces the concept of amortised optimisation and presented two case studies: optimisation of JIT compilation parameters of `pypy` Python runtime, and optimisation of the block size of Blocked Matrix Multiplication (BMM) algorithm. In both cases, the optimisation gradually takes place while the Software Under Metaheuristic Optimisation (SUMO) operates normally. Both implementations are available as ready-to-use open source libraries. Using these libraries, developers can inject online adaptivity into their software system, allowing users to gain performance simply by using the software repeatedly. The JIT optimisation can result in up to 8.6 % improvement in speed; the BMM optimisation can adapt to new hardware platform by finding an effective block size automatically.

Acknowledgement. We would like to thank Carl Friedrich Bolz and Laurence Tratt for the informative discussion about the technical details of `pypy`.

References

1. `CPUBoss`: a benchmark comparison between Xeon W3680 and Core-i7 3820QM. http://cpuboss.com/cpus/Intel-Xeon-W3680-vs-Intel-Core-i7-3820QM
2. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Inf. Softw. Technol. **51**(6), 957–976 (2009)
3. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: Pypy's tracing JIT compiler. In: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, pp. 18–25. ACM, New York (2009)
4. Briand, L.C., Labiche, Y., Shousha, M.: Stress testing real-time systems with genetic algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2005, pp. 1021–1028 (2005)
5. Budynek, J., Bonabeau, E., Shargel, B.: Evolving computer intrusion scripts for vulnerability assessment and log analysis. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2005, pp. 1905–1912 (2005)
6. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO 2005), pp. 1069–1075. ACM, Washington, D.C., 25–29 June 2005
7. Eves, H.: Elementary Matrix Theory. Dover Publication, New York (1980)
8. Groß, H.G.: An evaluation of dynamic, optimisation-based worst-case execution time analysis. In: ITPC 2003: Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu, pp. 8–14 (2003)
9. Grosso, C.D., Antoniol, G., Penta, M.D., Galinier, P., Merlo, E.: Improving network applications security: a new heuristic to generate stress testing data. In: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO 2005), pp. 1037–1043. ACM, Washington, D.C., 25–29 June 2005
10. Grosso, C.D., Antoniol, G., Merlo, E., Galinier, P.: Detecting buffer overflow via automatic test input data generation. Comput. Oper. Res. **35**(10), 3125–3143 (2008)
11. Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 612–621. IEEE Press, Piscataway (2013)
12. Kruse, P.M., Wegener, J., Wappler, S.: A highly configurable test system for evolutionary black-box testing of embedded systems. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2009, pp. 1545–1552 (2009)
13. Langdon, W., Harman, M.: Optimizing existing software with genetic programming. Trans. Evol. Comput. **19**(1), 118–135 (2015)
14. Penta, M.D., Canfora, G., Esposito, G., Mazza, V., Bruno, M.: Search-based testing of service level agreements. In: Proceedings of the Genetic and Evolutionary Computation, GECCO 2007, pp. 1090–1097 (2007)

15. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., García-Sánchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) EuroGP 2014. LNCS, vol. 8599, pp. 137–149. Springer, Heidelberg (2014)
16. Pohlheim, H., Wegener, J.: Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1795–1802, July 1999
17. Torres, M.: `Pypy` Speed Centre. http://speed.pypy.org/
18. Wegener, J., Grochtmann, M.: Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst. **15**(3), 275–298 (1998)
19. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: Genetic and Evolutionary Computation Conference (2015, to appear)