UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note

RN/12/13

NIA^{3}CIN: Non-Invasive Autonomous and Amortised Adaptivity Code Injection

November, 2012

Shin Yoo

Telephone: +44 (0)20 3108 5032
Fax: +44 (0)171 387 1397
Electronic Mail: s.yoo@cs.ucl.ac.uk
URL: http://www.cs.ucl.ac.uk/staff/S.Yoo

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 results in two problems. First, systems optimised in offline environment may not perform as well online. Second, the system needs to be taken offline and re-optimised when the environment changes. This paper introduces the novel concept of amortised optimisation to solve both problems, and presents an open source implementation. We evaluate the framework to optimise block matrix multiplication algorithm.

1 Introduction

Non-functional properties have been the focus of many Search Based Software Engineering (SBSE) work [1]. 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 considerable amount of interests [2–5]; other properties like Quality of Service [6, 7] and security [8–10] are emerging fields of research.

Most of 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, 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. This raises two issues about the optimality of the resulting solutions:

- **Sampling Bias:** It is often practically infeasible to capture the precise behaviour of the production environment, which can be infinitely varied (e.g. different combinations of hardware components) or not reproducible in development stage (e.g. realistic user load for web applications). As a result, offline approach will introduce bias towards the limited environment used for optimisation.
- **Lack of Persistent Adaptivity:** Even when the result of the initial optimisation meets the expectation, we cannot rule out the possibility of changes in the production environment that degrade the non-functional behaviour of the deployed system (e.g. upgraded hardware with different characteristics). As a result, the system should be taken off the production environment and re-optimised.

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 the 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 changed environment; the system will continue to adapt to changes.

We present an SBSE programming library called NIA^{3}CIN: Non-Invasive, Autonomous, and Amortised Adaptivity Code Injection. NIA^{3}CIN introduces a new approach to meta-heuristic optimisation, which is amortised optimisation. It makes a set of design choices, to deal with common concerns over introducing optimisation-based adaptivity into production stage software system. Following is a list of commonly expected questions and our responses:

- **Will it make my system slow?** No. NIA^{3}CIN does not run computationally expensive optimisation alongside your system. The optimisation cost is amortised: each run of your system serves as one fitness evaluation. What little overhead that NIA^{3}CIN introduces involves relatively simple bookkeeping (e.g. comparing the fitness of the neighbourhood for local search or performing genetic operations).
- **It must be hard to make my system self-adaptive.** No. NIA^{3}CIN is designed to be as non-invasive as possible. As long as the developer is aware of how to measure the property to be optimised, using NIA^{3}CIN involves only a couple of code annotations and a few method calls.
- **Can it make my system do wrong things?** No - as long as the developer does not use NIA^{3}CIN to optimise functional behaviours (e.g. finding the *correct* constant that is critical to the functional correctness of the computation). NIA^{3}CIN will operate within the search space defined by the developer, and can be turned off entirely.

As a proof of concept, we evaluated NIA^{3}CIN with a simple block matrix multiplication algorithm, whose ideal block size depends on the hardware details. We executed the algorithm, equipped with NIA^{3}CIN, and optimised the block size. Results show that NIA^{3}CIN can produce significant performance improvement without human intervention.

The rest of the paper is structured as follows. Section 2 introduces the concept of amortised optimisation. Section 3 discusses the internal implementation of NIA^{3}CIN, while Section 4 goes through a case study. Section 5 discusses related work. Section 6 concludes with the outlook for future work.

2 Amortised Optimisation

2.1 State-Based Approach

To achieve the design goals outlined in Section 1, we suggest amortising the execution of meta-heuristic optimisation so that the optimisation can take place *in situ*. Each execution of the target system (that requires optimisation) serves as a single fitness evaluation, which is similar to many traditional optimisation approaches. However, instead of having a tight optimisation loop wrapped around the fitness evaluation (i.e. the execution of the target system), NIA^{3}CIN performs only one step of the loop and stores the current status in the persistence layer. Consider the optimisation loop as a state transition system: whenever the target system requires to use the value to be optimised, NIA^{3}CIN is invoked and executes one or more state transitions, based on the current status of the optimisation. The returned value is used for the execution of the target system, which at the same time serves as a fitness evaluation for NIA^{3}CIN. Global states and other information, such as the number of remaining fitness evaluations, are stored in a separate persistence layer and loaded each time NIA^{3}CIN is invoked.

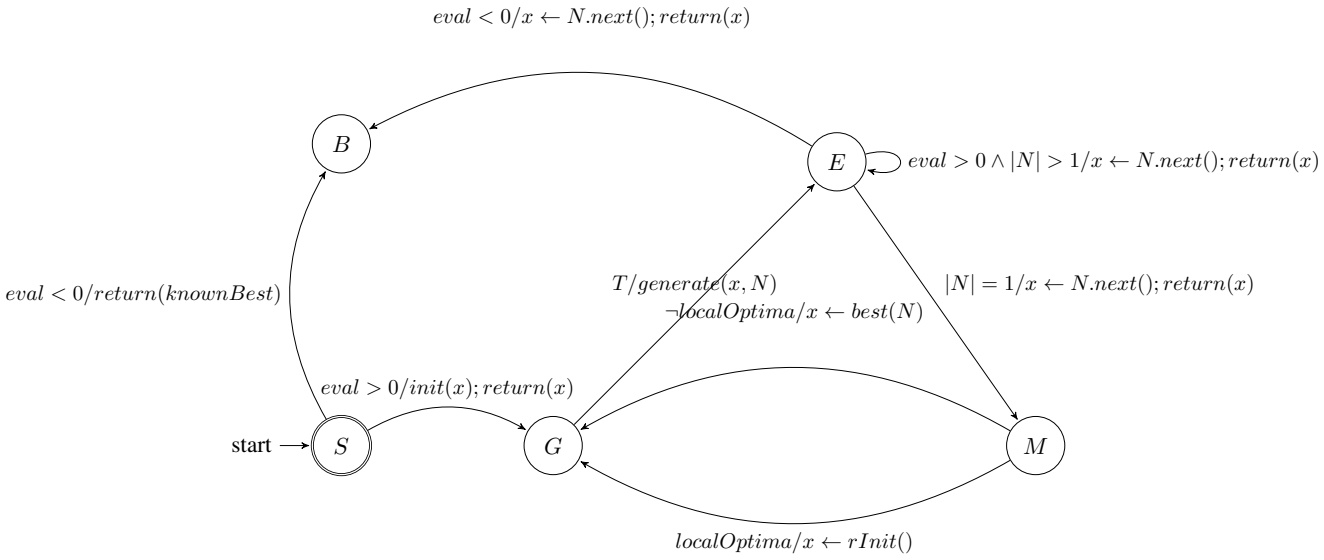


Figure 1: State-based model of amortised hill climbing algorithm: $return(x)$ decreases the remaining number of fitness evaluations, $eval$, by 1.

2.2 An Example: Hill Climbing

This paper presents an amortised version of hill climbing as a case study. Figure 1 is the outline of the state-based amortisation. Each state is defined as follows:

- **S**tart: When NIA^{3}CIN is invoked for the first time for the variable x, this is the starting state. If the remaining number of fitness evaluations, $eval$, read from the persistence layer is greater than 0, then NIA^{3}CIN sets x to the initial value given by the developer, returns it, and marks the next state as **G**.

- **G**enerate neighbours: This is the intermediate state that populates the neighbours queue, N, with the neighbouring solution of the current x. Once the queue is populated, NIA^{3}CIN executes the transition to **E**.
- **E**valuate: While $eval > 0$ and N contains more than 1 candidate solutions, NIA^{3}CIN returns the next neighbouring candidate solution. When the state reaches the final candidate solution in N, NIA^{3}CIN returns it and marks **M** as the next state.
- **M**ove: This is the state that is reached after evaluating all neighbouring candidate solutions in N. If the optimisation has reached a local optimum, NIA^{3}CIN applies a random restart to x and moves to **G**. If the current position is not a local optimum, then NIA^{3}CIN sets x to the best neighbouring solution and moves to **G**. From **G**, the next iteration of the hill climbing algorithm starts.
- **B**est: NIA^{3}CIN stores the best solution observed so far in the persistence layer. Whenever it runs out of allocated fitness evaluations, it will fall back to returning the known best value of x. Note that the fall-back transition only happens from **E** and **S**, i.e., NIA^{3}CIN can go over budget to finish one complete iteration of hill climbing.

3 Implementation & Usage

An implementation of NIA^{3}CIN is available at `https://bitbucket.org/ntrolls/niacin/` as an open source project. Currently the project supports hill climbing algorithm for Java; it also contains the case study that this paper considers. For an annotated variable (your input representation), NIA^{3}CIN will search for a value such that either maximise or minimise the value returned by another annotated method (your fitness function). Current implementation of NIA^{3}CIN uses Java annotation to interface user code non-invasively with optimisation code. Following is the list of the modifications required by the users of NIA^{3}CIN:

1. Mark the setter method for the input variable to the optimisation (i.e. the phenotype) with `@Input` annotation: this is the method that sets the value of the variable that NIA^{3}CIN will try to opmitise by searching for its value.
2. Write a method that returns a measure of your optimisation (i.e. the fitness function). For example, this method can return the time that an execution of a particular method has taken, or the memory that a particular data structure has consumed. Mark the method with `@Optimize` annotation.
3. When creating an instance of a class for which you want to apply NIA^{3}CIN, call `Niacin.initialize()` to initialise NIA^{3}CIN.
4. Immediately before the section of code you are interested, call `Niacin.start(this)`: this is the point when NIA^{3}CIN injects a new value for the variable using the setter method annoated with `@Input`.
5. Immediately after the section of code you are interested, call `Niacin.end(this)`: this is the poin when NIA^{3}CIN calls the method annotated with `@Optimize` to obtain the fitness value for this run.

The annotations allow a few basic properties to be defined for the methods:

- `@Input` annotation supports:
  - `name`: the name describing this variable
  - `initvalue`: the initial value to be used with this variable
  - `stepvalue`: the step increase/devrease amount when changing this variable (this is for local search heuristics such as hill climbing)

  - `bound`: the minimum and the maximum value for this variable, in the format of `"min,max"`.

- `@Optimize` annotation supports:
  - `direction`: either `Optimize.Direction.MIN` for minimisation or `Optimize.Direction.MAX` for maximisation.
  - `type`: the return type of the fitness method
  - `name`: name of the fitness value (`"fitness"`).
  - `trials`: the number of fitness evaluation allowed for this optimisation (default is 100).
  - `method`: the type of meta-heuristic algorithm to be used with this optimisation (default is `Metaheuristic.Type.HC`).

4 Case Study

4.1 Blocked Matrix Multplication (BMM)

As a proof of concept, we applied NIA^{3}CIN to tune parameters for blocked multiplication algorithm for square matrices, which is shown in Algorithm 1 and 2. Algorithm 1 breaks down the matrices into blocks, and invokes Algorithm 2 for each of them. This takes advantage of well known compiler optimisation techniques: having nested loops around a smaller region of memory locations exploits pipelining and caching to speed up the computation.

Algorithm 1: Blocked Matrix Multiplication
Input: Size of matrices, n, n-by-n matrices A and B
Output: matrix C, which equals to $A \cdot B$

(1) $n\_blocks \leftarrow \lceil \frac{n}{BLOCK\_SIZE} \rceil$
(2) **for** $b_i = 0$ **to** $b_i < n\_blocks$
(3) $i \leftarrow b_i * BLOCK\_SIZE$
(4) **for** $b_j = 0$ **to** $b_j < n\_blocks$
(5) $j \leftarrow b_j * BLOCK\_SIZE$
(6) **for** $b_k = 0$ **to** $b_k < n\_blocks$
(7) $k \leftarrow b_k * BLOCK\_SIZE$
(8) DO_BLOCK(n, A, B, C, i, j, k)

Algorithm 2: DO_BLOCK
Input: Matrix size, n, matrices A, B, and C, block indeces i, j, and k
Output: Updates matrix C

(1) $\_M \leftarrow (i + BLOCK\_SIZE > n?n - i : BLOCK\_SIZE)$
(2) $\_N \leftarrow (j + BLOCK\_SIZE > n?n - j : BLOCK\_SIZE)$
(3) $\_K \leftarrow (k + BLOCK\_SIZE > n?n - k : BLOCK\_SIZE)$
(4) **for** $\_i = 0$ **to** $\_i < \_M$
(5) **for** $\_j = 0$ **to** $\_j < \_N$
(6) $\_cij \leftarrow C[j + i * lda +_j +_i * lda]$
(7) **for** $\_k = 0$ **to** $\_k < \_K$
(8) $\_cij += A[i \cdot lda + k + \_i \cdot lda + \_k] \cdot B[j + k \cdot lda + \_j + \_k \cdot lda]$
(9) $C[j + i \cdot lda + \_j + \_i \cdot lda] = \_cij$

The key to the increased performance is the size of the block. However, this information depends on details of the hardware environment such as the cache size of the CPU. Hardcoding a fixed block size may produce optimal performance on one machine, but if the code is moved and executed on another machine, there is no guarantee that the same performance will be retained.

4.2 Applying NIA^{3}CIN to BMM

We discuss sections of blocked matrix multiplication here: the complete source code is available from the online repository [11]. First, we initialise NIA^{3}CIN when we create the instance of a BMM class:

```
public BlockedMatrixMultiplication()
{
  Niacin.initialise(BlockedMatrixMultiplication.class);
}
```

The variable we want to optimise with is the size of the block. We annotate the setter method as following:

```
@Input(name = "block_size", initvalue = "8", bound = "1, 128")
public void setBlockSize(int size)
{
  this.BLOCK_SIZE = size;
}
```

For fitness, we use the number of floating point multiplications we do per millisecond.

```
@Optimize(name = "rate", type = Double.class, direction = Optimize
    .Direction.MAX)
public Double getRate()
{
  return new Double(rate);
}
```

Now let us look at the actual method that implements Algorithm 1. At the beginning, we notify NIA^{3}CIN to inject the new value to be tried. We also mark the system clock time when this method begins.

```
public void square_dgemm(int M, int N, int K, double[] A, double[]
    B, double[] C)
{
  Niacin.start(this);
  long start = System.currentTimeMillis();
  ...
```

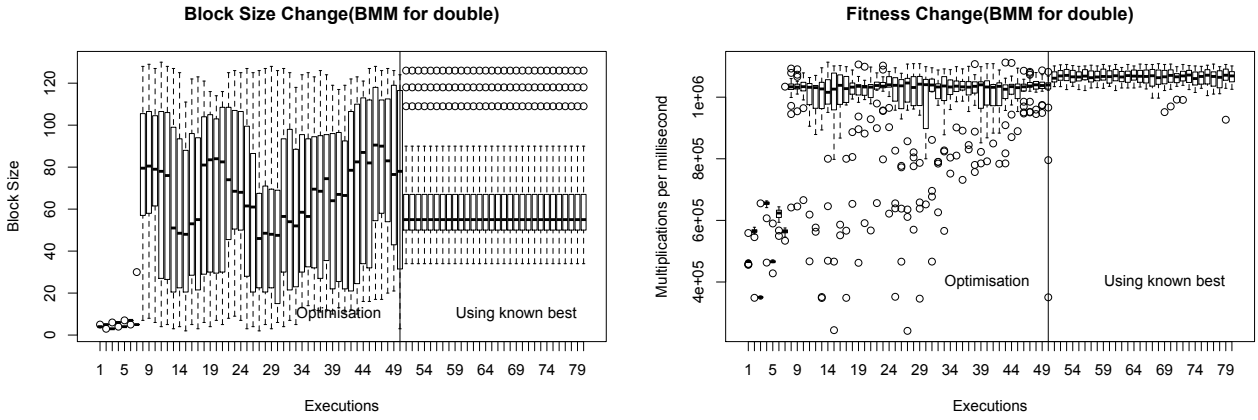
At the end of the same method, we first calculate our fitness, which is the number of floating point multiplications performed per millisecond. Then we notify NIA^{3}CIN that the fitness value is ready to be read back.

```
  ...
  long elapsed = System.currentTimeMillis() - start;
  rate = ((double) M * N * K) / (double) elapsed;

  Niacin.end(this);
}
```

4.3 Results

We have applied NIA^{3}CIN to an implementations of BMM for matrices of **double** type. Figure 2 shows the results from 20 runs of BMM for **double**. A *run* is 80 consecutive executions of BMM algorithm with NIA^{3}CIN, but NIA^{3}CIN has the budget of only only 50 fitness evaluations. After spending the first 50 executions, NIA^{3}CIN should fall back to the known best value. Figure 2(a) is a boxplot that shows how the block size changes throughout a run, across 20 runs. Since all runs start with the same initial block size



(a) (b)

Figure 2: Changes in block size and fitness (i.e. number of multiplications per millisecond) from 20 runs of BMM for **`double`**. Each run consists of 80 executions of BMM; the first 50 run uses NIA^{3}CIN to optimise the block size, after which it uses the best size known so far.

of 4, the block size remains small until hill climbing finds the locam optima. Random resets follow, which explains the large variance in the middle section. Eventually each run decides on a value. Figure 2(b) is a boxplot that shows how the fitness changes througout a run, also across the same 20 runs. Once hill climbing starts random resets, fitness also starts varying but multiple convergences take place throughout the optimisation stage. On average, the performance improved by more then two times.

5 Related Work

Langdon and Harman improved non-functional property of a non-trivial C++ program using Genetic Programming (GP) [12]. 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. As far as we know, this is the first time the optimisation of a non-functional behaviour is completely amortised and injected into the target system.

6 Conclusion and Future Work

This paper introduces the concept of amortised optimisation and presented an implementation called NIA^{3}CIN. We observe that any optimisation loop can be reconstructed as a state-based model, which can be executed step by step: this allows us to amortise the execution of the optimisation itself. We presented a case study that requires hardware dependent parameter tuning, for which NIA^{3}CIN successfully delivers performance improvement.

Future work include more serious evaluation of NIA^{3}CIN approach as well as extension of the technique. The more frequently a program is used, the more benefit we can harvest by making it more efficient. At the same time, the more frequently a program is used, the more fitness evaluation it provides for NIA^{3}CIN, making the optimisation stronger. We note that cloud computing, where many users execute the same software system in a large scale, fits this profile very well. The extension will mainly consider two things. First, NIA^{3}CIN can only handle side-effect free variables whose value will may affect non-functional properties but not the functional correctness. Handling variables that can potentially affect functional correctness will require more careful approach. Second, concentrated and highly connected server farms for cloud computing will provide an ideal ground for population-based optimisation; here, each machine

will asynchronously perform fitness evaluations, results of which can be either shared in peer to peer fashion, or by a central host that controls the entire optimisation.

References

[1] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[2] H. Pohlheim and J. Wegener, "Testing the temporal behavior of real-time software modules using extended evolutionary algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*. Morgan Kaufmann, July 1999, pp. 1795–1802.

[3] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, no. 3, pp. 275 – 298, November 1998.

[4] H.-G. Groß, "An evaluation of dynamic, optimisation-based worst-case execution time analysis," in *ITPC'03: Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu*, 2003, pp. 8–14.

[5] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1021–1028.

[6] M. D. Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based testing of service level agreements," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. London, England: ACM, 7-11 July 2007, pp. 1090–1097.

[7] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*. Washington, D.C., USA: ACM, 25-29 June 2005, pp. 1069–1075.

[8] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers and Operations Research*, vol. 35, no. 10, pp. 3125–3143, October 2008.

[9] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo, "Improving network applications security: A new heuristic to generate stress testing data," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*. Washington, D.C., USA: ACM, 25-29 June 2005, pp. 1037–1043.

[10] J. Budynek, E. Bonabeau, and B. Shargel, "Evolving computer intrusion scripts for vulnerability assessment and log analysis," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1905–1912. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068331

[11] NIA^{3}CIN: Non-Invasive Autonomous and Amortised Adaptivity Code Injection. code repository: http://bitbucket.org/ntrolls/niacin.

[12] W. B. Langdon and M. Harman, "Genetically improving 50,000 lines of C++," Department of Computer Science, University College London, Tech. Rep. RN/12/09, 2012.