Amortised Optimisation as a Means to Achieve Genetic Improvement

Hyeongjun Cho, Sungwon Cho, Seongmin Lee, Jeongju Sohn, and Shin Yoo
Offline Improvement

We modified two C++ files: Solver.cc, containing the core solving algorithm (321 out of 582 lines of code), and SimpSolver.cc, which simplifies the input instance (327 out of 480 lines of code).

Furthermore, we were evolving a list of changes, that is, a list of copy, replace and delete instructions. We only kept such lists in memory, instead of multiple copies of an evolved source code.

For each generation the top half of the population was selected. These were either mutated, by adding some of the three operations mentioned above, or crossover was applied, which simply merged two lists of changes together. Mutation and crossover took place with 50% probability each. New individuals were created by selecting one of the three mutation operations.

For each generation five problems were randomly chosen from the five groups of test cases. Fitness was evaluated as follows: if correct answer was returned by an individual, 2 points were added; if, additionally, the modified program was faster, 1 more point was added. Only individuals with 10 or more points were considered for selection. In order to avoid environmental factors, we counted the number of lines used to establish whether a mutated program was more efficient than the original one. The whole process is presented in Figure 1.

Fig. 1. GP improvement of MiniSAT.

4 Initial Results

A summary of our results is shown in Table 1. We refer to versions of MiniSAT that run faster than the unmodified solver on the maximum set of instances as 'best individuals'.

We ran our experiments on a test suite with 71 test cases taken from the 2011 SAT competition. Each generation contained 20 individuals. Time limit was set to 25 seconds and it took 14 hours to produce 100 generations. We only modified the Solver.cc file, containing the core solving algorithm. Of all programs generated 73% of them compiled. The best one was more efficient than the unmodified solver on 70 SAT instances, in terms of lines of code used. However, the modified versions mostly just removed assertions as well as some statistical data. Some optimisations have also been deleted, but these in turn led to longer runtimes on certain instances.

Next, we selected the test cases from only the application tracks of SAT competitions. MiniSAT was able to find an answer for 107 problems out of 500 instances tested.
Environmental Factors

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

One Generation

fitness evaluation

selection
crossover
mutation
...

One Generation
Amortised Optimisation

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

[Image: Keep Calm and Carry On]

Genetic Improvement, out in the wild!

Budget Controlled (will stop when run out)

[Image: Caution: Low Overhead Clearance]

Low Overhead (only microscopes)
Does it work?

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

When to begin tracing?

When to mark as hot?

When to compile the bridge?
PIACIN

1. Install the package.
2. Import the package
3. There is no step 3.
Table 1. Benchmark user scripts used for the JIT optimisation case study

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


$^b$ 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/
How about hardware?

Let us consider matrix multiplication.

Optimal block size depends on L1 size.

Blocked Matrix Multiplication: smaller inner loop to fit everything into L1 cache.
NIA\textsuperscript{3}CIN

Non-Invasive, Amortised and Autonomous Code Injection

Annotation-based

Event-driven dependency injection
Evaluation

4.2 Experimental Setup

Implementation

We use a Java implementation of the BMM algorithm for matrices of double type. The amortised optimisation framework, called Name X (Non-Invasive Amortised & Automated Adaptivity Code Injection), uses the hill climbing algorithm and is also implemented in Java. To be as little intrusive as possible, Name X 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 in second), need to be marked with annotation. Before the parameter is to be used, the SUMO needs to call Name X so that the parameter variable is updated with the current solution; after the parameter has been used, the SUMO needs to call Name X so that the fitness is fed back to the optimisation.

The range of block size was set to $[1, 512]$. Name X 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 Name X deliberately, providing opportunities to evaluate the same solution more than once (and, therefore, getting clearer measures of the fitness).

Table 3. Information about CPUs for which BMM was optimised

<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock Frequency</th>
<th>L1 Instruction Cache</th>
<th>L1 Data Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon W3680&lt;sup&gt;a&lt;/sup&gt;</td>
<td>3.33GHz</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>Intel Core-i7 3820QM&lt;sup&gt;a&lt;/sup&gt;</td>
<td>2.7GHz</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>ARM1176 (BCM2835 SoC)&lt;sup&gt;b&lt;/sup&gt;</td>
<td>250MHz</td>
<td>16KB</td>
<td>16KB</td>
</tr>
</tbody>
</table>

<sup>a</sup> These Intel CPUs share data and instruction caches between two processor threads.

<sup>b</sup> Raspberry Pi Model B, first edition.
GPGPU
Workgroup Size

- Local Workgroup Size: decides how many threads are executed by stream multiprocessor units
- Too small: under-utilised GPU
- Too large: local memory spill, resulting in costly I/O with RAM
Exposing hidden parameter: Deep Parameter Optimisation

- For cases where parameter that controls the performance is hidden
- Expose ‘deep’ (previously hidden) parameter to be explicitly controlled
- Our case,
  - Local work group size for GPGPU module of OpenCV controls the performance
  - Should be exposed to be explicitly controlled for optimisation of the performance

---

Exposing hidden parameter: Deep Parameter Optimisation

- For cases where parameter that controls the performance is hidden
- Expose ‘deep’ (previously hidden) parameter to be explicitly controlled
- Our case,
  - Local work group size for GPGPU module of OpenCV controls the performance
    ➔ Should be exposed to be explicitly controlled for optimisation of the performance

---

Results

Default  Amortised optimisation  Best

Match 3

Execution time (s)

0.15  0.25  0.35

Exuctions

Match 20

Execution time (s)

0.35  0.45
Web servers run in many devices: Raspberry Pi, rack servers, desktop PCs, …

But they have the same Apache2 parameters!
Methodology – Objective / Fitness

- Server Side (unit: %):
  - Minimize (average CPU usage) + (average memory usage)

- Client Side (unit: sec):
  - Minimize (max response time) / 10 + (average response time)

- Measured 2 times, use average value
Experiments

• Server Environments:
  • Xen Virtual Server (Hosted by SPARCS)
  • Target: a simple MediaWiki site (Apache2.4 + PHP5 + MySQL)
  • 1st Server (CPU: Xeon E5645@2.40GHz 1 core / Memory: 256MB)
  • 2nd Server (CPU: Xeon E5645@2.40GHz 2 cores / Memory: 2GB)

• Client Environments:
  • Sungwon’s Personal Computer: Ubuntu, Same Subnet
  • Microsoft Azure: Ubuntu, Different Subnet
## Results

- **1st Server, 1st Scenario (around 60min):**

<table>
<thead>
<tr>
<th></th>
<th>CPU AVG</th>
<th>MEM AVG</th>
<th>TIME MAX</th>
<th>TIME AVG</th>
<th>SERVER</th>
<th>CLIENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>86.3445</td>
<td>61.6227</td>
<td>2.8777</td>
<td>0.8150</td>
<td>147.9672</td>
<td>1.1028</td>
</tr>
<tr>
<td>OUR SOL</td>
<td>82.2728</td>
<td>50.5601</td>
<td>1.7528</td>
<td>0.7327</td>
<td>132.8329</td>
<td>0.9080</td>
</tr>
</tbody>
</table>

- **1st Server, 2nd Scenario (around 70min):**

<table>
<thead>
<tr>
<th></th>
<th>CPU AVG</th>
<th>MEM AVG</th>
<th>TIME MAX</th>
<th>TIME AVG</th>
<th>SERVER</th>
<th>CLIENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>85.3299</td>
<td>66.2125</td>
<td>2.8559</td>
<td>0.8190</td>
<td>151.5424</td>
<td>1.1046</td>
</tr>
<tr>
<td>OUR SOL</td>
<td>85.5942</td>
<td>47.3762</td>
<td>1.3903</td>
<td>0.7653</td>
<td>132.9704</td>
<td>0.9043</td>
</tr>
</tbody>
</table>
Threats

Restricted to behaviour-preservation optimisations only

User may experience performance fluctuation

We want you!

Getting precise measurements
Next Steps

- Population-based optimisation using multiplicity: for example, swarm optimisation of performance-critical parameters in a data centre.

- Shadowing: parallel instance dedicated for optimisation.

- Prepackaged GI: GI as aspects, tagging, directives
References


Amortised Optimisation

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

Does it work?

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

Evaluation

Table 3. Information about CPUs for which BMM was optimised

<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock Frequency (GHz)</th>
<th>Instruction Cache L1</th>
<th>Data Cache L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7 Kaby</td>
<td>3.40</td>
<td>6MB</td>
<td>6MB</td>
</tr>
<tr>
<td>Intel Core i5 Skylake</td>
<td>3.20</td>
<td>6MB</td>
<td>6MB</td>
</tr>
<tr>
<td>ARM1176 (Broadcom SoC)</td>
<td>1.05</td>
<td>16KB</td>
<td>16KB</td>
</tr>
<tr>
<td>AR8125 (QCOM SoC)</td>
<td>1.05</td>
<td>16KB</td>
<td>16KB</td>
</tr>
</tbody>
</table>

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

GPGPU

Workgroup Size

- Local Workgroup Size: decides how many threads are executed by stream multiprocessor units
- Too small: under-utilised GPU
- Too large: local memory spill, resulting in costly I/O with RAM

Tuning MPM

Modules for Apache

- Web servers run in many devices: Raspberry Pi, rack servers, desktop PCs, ...
- But they have the same Apache2 parameters!

Code Available

https://bitbucket.org/ntrolls/piacin

https://bitbucket.org/ntrolls/niacin