



南開大學  
Nankai University

OS 功能挑战赛 proj105

---

yaTuner: 编译器自动调优

---

指导教师：李雨森

队伍名称：库乐队

队伍成员：梅骏逸 郭大玮

2022 年 8 月 15 日

# 目录

<b>一、 概述</b>	<b>1</b>
<b>二、 问题分析</b>	<b>1</b>
<b>三、 相关工作</b>	<b>2</b>
(一) 贝叶斯优化 . . . . .	2
(二) CompilerGym . . . . .	2
(三) Opentuner . . . . .	2
<b>四、 算法简述</b>	<b>2</b>
(一) 假设检验 . . . . .	2
1. 应用可行性 . . . . .	2
2. 应用于二元编译选项的假设检验算法 . . . . .	3
3. 应用于带参编译选项的假设检验算法 . . . . .	4
4. 假设检验算法的优势 . . . . .	5
(二) LinUCB . . . . .	5
1. 适用性 . . . . .	5
2. 特征的选取 . . . . .	6
3. 适用于带参编译选项的 LinUCB 算法 . . . . .	6
<b>五、 yaTuner 架构</b>	<b>7</b>
(一) 调优脚本的生成与配置 . . . . .	8
(二) 多目标调优支持 . . . . .	9
(三) 自动获取编译器优化选项 . . . . .	10
(四) 执行优化 . . . . .	10
(五) 生成报告 . . . . .	10
<b>六、 测试结果</b>	<b>13</b>
(一) PolyBench . . . . .	13
(二) 其它程序 . . . . .	15
<b>七、 总结</b>	<b>15</b>

## 一、 概述

编译器的编译选项对于程序的运行性能等指标会产生重大影响。事实上，现代高级语言的运行效率在很大程度上依赖于编译器的优化水平。所以选择编译时的选项就显得尤为重要。而对于不同的场景，有着不同的优化目标的需求，而与此有关的选项的选择很大程度上有赖于经验。yaTuner 使用假设检验、贝叶斯优化以及 LinUCB 算法，实现了对多目标的自动调优，并且取得了不错的效果<sup>1</sup>。

## 二、 问题分析

编译器优化选项的选择是一个典型的黑盒优化问题，此类问题已经得到了广泛的研究，有多种不同的数学模型和优化算法可以使用，但由于编译器优化问题的特殊性，很多优化算法无法直接应用，主要有以下几个关键问题：

### • 优化选项种类各异

主流编译器（如 GCC、LLVM 等）通常提供了 Optimizers（如 `-fast-math`）之类的二元编译选项（即只能选择是否开启），Parameters（如 `hot-bb-frequency`）之类的带参数编译选项，以及离散选项（如 `parloops-schedule=[static|dynamic|guided|auto|runtime]`）单一优化方法很难在不同类型选项中获得最优解，需要多种优化方法结合优化。此外不同的选项对工具的接口设计和鲁棒性有较大的考验。

### • 搜索空间庞大

主流编译器提供的编译选项数量庞大（如 GCC10.3 提供了 230 个 Optimizers 和 225 个 Parameters），如直接对所有的选项进行优化，仅 Optimizers 的搜索空间便可达到  $2^{230}$ ，总维度可达近 500 维，传统优化方法在如此大的维度下很难得到较好的结果。而有效优化选项又与程序本身息息相关，传统手动筛选的方式不仅需要大量经验的积累，同时也很容易漏选错选，往往会陷入选择过多优化时间过久甚至无法计算，选择过少优化效果又不好的两难境地。

### • 某些优化选项会导致编译器报错

某些优化选项的正确编译高度依赖程序编写的标准性和严谨性，甚至某些较为激进的优化选项对程序有着极尽苛责的要求，编译器在进行某些优化时遇到不合规、包含未定义行为的程序，会无法编译导致报错，这就对自动优化算法的错误处理能力和鲁棒性有了较高的要求。

### • 程序运行时间具有不确定性

经过前期调研和实验分析，我们发现受系统调度、分支预测影响，程序的运行时间往往具有较大的随机性，已有的自动调优方案往往仅进行一次采样，没有对运行时间的不确定性进行考虑，会导致优化算法受异常偏小值影响，无法得到最优结果。同时可能会受到系统性能波动影响，造成统计误差。

---

<sup>1</sup>本文档所有数据的获取过程、调优脚本的源代码均可以在项目仓库中找到，但是由于不同平台性能、编译器版本与优化能力以及调优运行过程中程序运行时间的不确定性，实际运行结果可能与文档中数据有所差异。

## 三、 相关工作

### (一) 贝叶斯优化

贝叶斯优化是一种高效的超参数优化手段，常用于解决黑盒优化问题。但在维度较高的情况下，贝叶斯优化的计算会变得极其困难，且由于程序运行时间不恒定，观测结果的噪声较大，传统贝叶斯优化无法处理这种高水平的噪声。

### (二) CompilerGym

CompilerGym [3] 提供了一个用于编译有关任务的强化学习环境，并对编译输出的特定文件提供了指令数量、特征等有关的指标用于强化学习过程。对于 LLVM IR 来说，每一次优化所生成的新的中间表示（即 IR）的确可以作为一个决策过程的结果并通过强化学习方法进行进一步优化。但是对于 GCC 以及其它并非使用 LLVM IR 的编译器来说，其编译生成的结果多样，在不同场景下的编译过程也是不同的，此时优化的过程便难以直接将迭代优化 LLVM IR 的方法应用到上面。

### (三) Opentuner

Opentuner [1] 实现了包括差分进化（Differential Evolution）、Nelder-Mead 搜索、粒子群优化、bandit 变异技术等一系列混合算法进行优化，但其优化过程随机性强，结果不稳定。

## 四、 算法简述

为了解决上述的问题，我们将假设检验的方法和 LinUCB 算法应用到了编译器的自动调优上。

### (一) 假设检验

#### 1. 应用可行性

在前期调研过程中，我们发现程序运行的时间具有较强的随机性，通过对一次编译得到的程序多次运行，我们能够绘制出运行时间频数分布如图1。

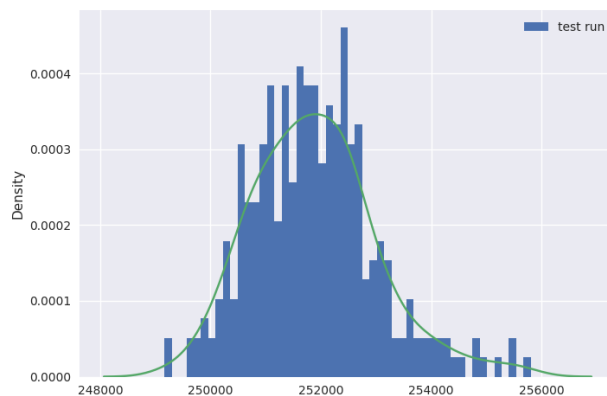


图 1: 运行时间频数分布图

通过观察,我们发现程序运行的时间近似符合正态分布,依此,我们使用 K-S Test 对数据进行检验,验证了程序运行时间近似正态分布<sup>2</sup>。

由此,我们便可以通过假设检验的方式验证两次编译得到的程序运行速度是否有明显差异,相比单次运行后比较运行时间,通过假设检验验证的结论更可靠,在一定程度上能够避免偶然因素造成的误差。

## 2. 应用于二元编译选项的假设检验算法

对于 Optimizers 类的选项,其状态只有 01 两种,即开启与不开启,由此我们进行以下假设检验:

### 1. 建立假设和确定检验水准

零假设  $H_0$ : 开启该优化选项后的  $\mu \geq \mu_0$

备择假设  $H_1$ : 开启该优化选项后的  $\mu < \mu_0$

预先规定显著性水准 (significant level)  $\alpha$ , 文中数据选取  $\alpha = 0.05$

### 2. 使用学生 T 检验 (Student's T-Test) 计算检验统计量

由于程序运行时间往往较长,我们采取小样本检测 ( $n < 30$ , 文中数据选取  $n = 5$ ), 采用单样本学生 T 检验 (Student's T-Test) 进行单侧检验。

对打开该选项后得到的可执行文件多次运行进行采样 (文中数据选取  $n = 5$ ), 按照 T 检验算法计算统计量  $t$  值。

### 3. 确定 p 值, 做出推断结论

按照显著性水准  $\alpha$ , 单侧检验, 自由度  $df = n - 1$  查 T 检验临界值表, 确定 p 值。

若  $p \leq \alpha$ , 按检验水准  $\alpha$  拒绝  $H_0$ , 认为该选项对提升程序性能有明显帮助,

若  $p > \alpha$ , 不拒绝  $H_0$ , 认为该选项对提升程序性能没有明显帮助。

依此对 Optimizers 进行逐一检测, 便可筛选出对该程序敏感, 能够帮助提升程序性能的选项。

在实际优化流水线中, yaTuner 选用以下计算流程进行 Optimizers 的自动预筛选。

---

#### Algorithm 1: Hypothesis testing for optimizers

---

**Input:** optimizers set  $K$ ; threshold  $\alpha$ ; sample size  $m, n$ ;

**Output:** selected optimizers set  $S$ ;

$B \leftarrow$  sample baseline with size  $m$ ;

$\mu \leftarrow \bar{B}$ ;

**forall** optimizer  $k \in K$  **do**

$M \leftarrow$  sample with optimizer  $k$  on with size  $n$ ;

$t \leftarrow T\_test\_one\_sample(M, \mu)$ ;

**if**  $t < -t_{\alpha, n-1}$  (table of selected values for  $t$ -distribution, one sided) **then**  
        let  $k \in S$

**end**

**end**

---

<sup>2</sup>在实际调优过程中可能会出现程序运行时间不符合正态分布的情况, 此种情况可能由多种原因所致 (系统调度、样本数量太少等), 出于实际调优实践的考虑, 我们仍然将其视为正态分布处理, 并提供了截取及对称化的选项以进行粗略修正。

### 3. 应用于带参编译选项的假设检验算法

对于 Parameters 类的选项，其参数值介于编译器内部定义的区间  $[min, max]$  内，我们进行以下假设检验：

#### 1. 建立假设和确定检验水准

分别选择该选项参数最小值 (min)<sup>3</sup>和参数最大值 (max)<sup>4</sup>做为两组编译选项

零假设  $H_0$ : 两组编译选项  $\mu_1 = \mu_2$

备择假设  $H_1$ : 两组编译选项  $\mu_1 \neq \mu_2$

预先规定显著性水准 (significant level)  $\alpha$  , 文中数据选取  $\alpha = 0.05$

#### 2. 使用学生 T 检验 (Student's T-Test) 计算检验统计量

采取小样本检测 ( $n < 30$ , 文中数据选取  $n = 5$ ), 采用学生 T 检验 (Student's T-Test) 进行两独立样本均值检验。

对两编译选项得到的可执行文件分别多次运行进行采样 (文中数据选取  $n = 5$ ), 先对两样本进行 levene 方差齐性检验 (Levene test), 判断是否符合方差齐性, 之后按照两独立样本 T 检验算法计算统计量  $t$  值。

#### 3. 确定 p 值, 做出推断结论

按照显著性水准  $\alpha$ , 双侧检验, 依照方差齐性选取  $n - 1$  或按照 Satterthwaite 近似法计算自由度  $df$  查 T 检验临界值表, 确定 p 值。

若  $p \leq \alpha$ , 按检验水准  $\alpha$  拒绝  $H_0$ , 认为该选项对程序性能有显著影响,

若  $p > \alpha$ , 不拒绝  $H_0$ , 认为该选项对提升程序性能没有显著影响。

依此对 Parameters 进行逐一检测, 便可筛选出对该程序敏感, 能够影响程序性能的选项。在实际优化流水线中, yaTuner 选用以下计算流程进行 Parameters 的自动预筛选。

---

#### Algorithm 2: Hypothesis testing for parameters

---

**Input:** parameters set  $K(name, min, max)$ ; threshold  $\alpha$ ; sample size  $n$ ;

**Output:** selected parameters set  $S$ ;

**forall** parameter  $k \in K$  **do**

$M_{min} \leftarrow$  sample with optimizer  $k_{min}$  with size  $n$ ;

$M_{max} \leftarrow$  sample with optimizer  $k_{max}$  with size  $n$ ;

**if**  $Levene\_test(M_{min}, M_{max}) < \alpha$  **then**

$t \leftarrow T\_test\_equal\_variance(M_{min}, M_{max})$ ;

**else**

$t \leftarrow T\_test\_unequal\_variance(M_{min}, M_{max})$ ;

**end**

**if**  $|t| > t_{\alpha/2, v}$  (table of selected values for  $t$ -distribution, two sided) **then**

        let  $k \in S$ ;

**end**

**end**

---

<sup>3</sup>因某些编译选项参数为 0 时与参数较大时等效, 推荐使用最小值 +1 作为参数最小值。

<sup>4</sup>某些选项并没有提供范围的获取, 对于此类选项, 我们通过获取默认值方法将其最大值设为十倍默认值。

#### 4. 假设检验算法的优势

- **考虑了程序运行时间的随机性**

采用假设检验的方式考虑了变量的随机性，可以判断样本与样本、样本与总体的差异是由抽样误差引起的还是本质差别造成的，将犯弃真错误的概率（即该选项对于提升无益而被选择的概率）限制在可以接受的范围内（一般为 5%），减少了采样误差对程序的影响。

- **自动降维**

通过假设检验算法，yaTuner 实现了对编译选项的自动预筛选，大大降低了搜索维度，显著提升了优化效率，同时自动化筛选过程能够解决传统手动筛选依赖大量经验，过程复杂的问题，降低了工具的使用门槛。

经过测试假设检验算法可以在 GCC 编译器的选项自动筛选上取得显著的效果，可以将编译选项从几百个降低为数十个。下表展示了对一个测试程序的选项进行筛选的结果。

	Before	After	Percentage
Optimizers <sup>5</sup>	81	12	14.8%
Parameters	227	6	2.6%

表 1: 假设检验预筛选前后的优化选项数<sup>6</sup>

- **减少计算开销**

假设检验算法的计算开销极小，取得的效果显著（见表3 HypoT 栏），相比于各种搜索算法在庞大搜索空间中的搜索效率，假设检验方法计算量小，结果准确，同时极大地减小了后续搜索的搜索空间，减少了后续优化的计算量。

- **稳定性高**

基于假设检验的方法对选项的筛选效果非常稳定，能够在固定时间内准确地筛选出对该程序有效的编译选项，克服了搜索类算法随机性强，难以判断是否收敛和优化终止条件的问题。

## （二） LinUCB

### 1. 适用性

LinUCB [4] 是一种基于上下文 (context-based) 和 UCB 的 Bandit 算法，其通过假设 reward 与特征 (feature) 具有线性关系，用特征预估回报及其置信区间，选择置信区间上界最大的 arm，观察回报后更新线性关系的参数，以此达到试验学习的目的。

在编译器优化问题中（例如对程序运行时间的优化），可以获取程序运行时或静态的特征，将其结果（如性能计数器）作为 context，采用 LinUCB 算法学习编译选项与 context 的关系，可以加快收敛速度，减少优化时间。同时 UCB 的实现考虑了程序运行时间的随机性，避免了采样误差对算法的影响。

<sup>5</sup>在 O3 基础上优化，预先去掉了 O3 已经开启的选项。

<sup>6</sup>测试程序为 Triple，结合了 Matmul, RayTracer, TSP-GA 三个程序，这三个程序均来自于 Opentuner 对 g++ 的调优示例，编译器为 GCC for openEuler 10.3.1。

## 2. 特征的选取

yaTuner 内置了对 linux perf 性能分析工具的调用功能<sup>7</sup>, 通过执行 perf 命令读取其输出并进行处理获取结果。yaTuner 内置的工具选取了以下 counters 作为特征:

Counters				
branch-instructions	alignment-faults	cpu-cycles	emulation-faults	cache-references
branch-misses	bpf-output	instructions	major-faults	cpu-migrations
bus-cycles	context-switches	ref-cycles	minor-faults	duration_time
cache-misses	cpu-clock	task-clock	page-faults	

表 2: yaTuner 内置工具使用的 perf counters

## 3. 适用于带参编译选项的 LinUCB 算法

yaTuner 对需要优化的 Parameters 进行离散化处理后, 对每个 Parameter 构建一个 LinUCB, 采用随机化方法, 使多个 LinUCB 进行并行优化, 每次迭代中对选用的编译选项进行编译运行, 获取运行时间和性能计数器, 以性能计数器作为 context, 以  $t_0 - t_i$  作为 reward, 迭代训练 LinUCB 直至收敛。最后选取最佳一次运行结果作为搜索结果。

<sup>7</sup>这一工具属于 `yatuner.utils` 模块, 旨在简化调优脚本的编写过程而非强制使用, 对于不同场景的不同特征分析方式, 仍然可以通过调整生成的初始调优脚本进行实现。



**Algorithm 3:** Parallel LinUCB for parameters

---

**Input:** parameters set  $K(param, min, max)$ ; number of bins  $M$ ;  
 random selection size  $N$ ;  $\alpha \in \mathbb{R}_+$

**Output:** optimized parameters set  $S(param, val)$ ;

**forall** parameter  $k \in K$  **do**  
 Initialize  $\mathcal{A}_k$  with arms of  $M$  evenly distributed values  $\in [k_{min}, k_{max}]$ ;  
**forall**  $a \in \mathcal{A}_k$  **do**  
 $\mathbf{A}_{k,a} \leftarrow \mathbf{I}_d$  ( $d$ -dimensional identity matrix);  
 $\mathbf{b}_{k,a} \leftarrow \mathbf{0}_{d \times 1}$  ( $d$ -dimensional zero vector);  
**end**  
**end**

**for**  $t = 1, 2, 3, \dots, T$  **do**  
**forall** parameter  $k \in K$  **do**  
**forall**  $a \in \mathcal{A}_k$  **do**  
 $\hat{\boldsymbol{\theta}}_{k,a} \leftarrow \mathbf{A}_{k,a}^{-1} \mathbf{b}_{k,a}$ ;  
 $p_{k,a} \leftarrow \hat{\boldsymbol{\theta}}_{k,a}^T \mathbf{x}_{k,a} + \alpha \sqrt{\mathbf{x}_{k,a}^T \mathbf{A}_{k,a}^{-1} \mathbf{x}_{k,a}}$ ;  
**end**  
 Randomly choose arm  $a_k$  from  $N$  arms with highest  $p_{k,a}$ ;  
 $S_k \leftarrow a_k$ ;  
**end**  
 Observe performance features  $\mathbf{x}_t$  with current parameters  $S$  and observe payoff  $r$ ;  
**forall** parameter  $k \in K$  **do**  
 $\mathbf{A}_{k,a_k} \leftarrow \mathbf{A}_{k,a_k} + \mathbf{x}_{k,a_k} \mathbf{x}_{k,a_k}^T$ ;  
 $\mathbf{b}_{k,a_k} \leftarrow \mathbf{b}_{k,a_k} + r \mathbf{x}_{k,a_k}$ ;  
**end**  
**end**  
 Choose  $S$  with highest reward;

---

## 五、 yaTuner 架构

yaTuner 的总体架构如图1所示。

yaTuner 使用 Python 语言实现，通过自动生成基于 Python 语言的调优脚本文件并对其进行手动微调以适应不同场景的需求。运行调优脚本后能够在指定目录下存储有关调优结果以及可视化图表等。

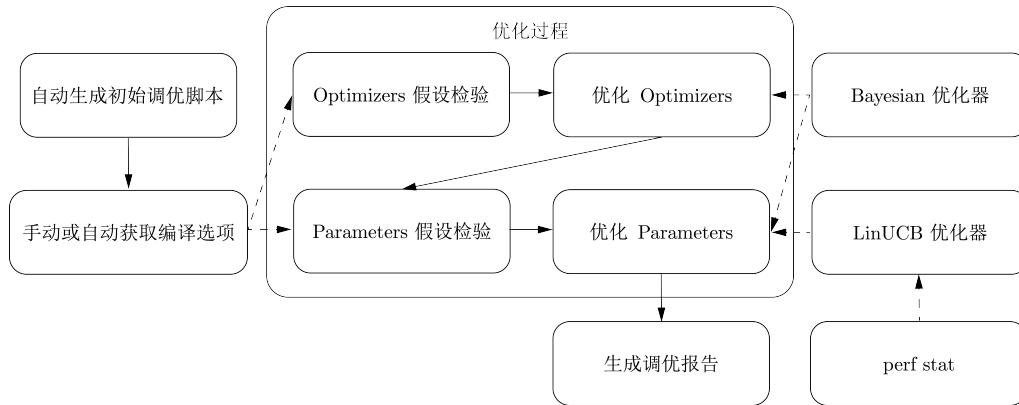


图 2: yaTuner 运行架构图

### (一) 调优脚本的生成与配置

yaTuner 支持预设调优脚本的自动生成，基于自动生成的调优脚本（如下<sup>8</sup>），结合 yaTuner 提供的内置工具，可以便捷地对脚本进行进一步的细化配置。

```

1  import logging
2  import yatuner
3  import os
4
5  cc = 'gcc'
6  build_dir = './build'
7  workspace = './workspace'
8  metric = 'duration_time'
9
10 if not os.path.isdir(build_dir):
11     os.mkdir(build_dir)
12
13 optimizers = yatuner.utils.fetch_gcc_optimizers(cc=cc)
14 parameters = yatuner.utils.fetch_gcc_parameters(cc=cc)
15
16 def comp(optimizers, parameters, additional):
17     raise NotImplementedError("You need to tell yatuner how to compile.")
18
19 def run():
20     raise NotImplementedError("You need to tell yatuner how to run.")
21
22 def perf():
23     raise NotImplementedError()
24
25 tuner = yatuner.Tuner(call_compile=comp,
26                       call_running=run,
27                       call_perf=perf,
28                       optimizers=optimizers,
29                       parameters=parameters,
30                       workspace=workspace,
31                       log_level=logging.INFO,
32                       norm_range=1.0)
33 tuner.initialize()

```

<sup>8</sup>有所删减。

```

34 tuner.test_run(num_samples=500, warmup=0)
35 tuner.hypotest_optimizers(num_samples=5)
36 tuner.hypotest_parameters(num_samples=5)
37 # tuner.optimize(num_samples=10, num_epochs=50) # using bayesian optimization
38 tuner.optimize_linUCB(alpha=0.25,
39                       num_bins=30,
40                       num_epochs=200,
41                       nth_choice=4,
42                       metric=metric) # using linUCB
43 tuner.run(num_samples=50)
44 tuner.plot_data()

```

一般来说，对于 yaTuner 初始生成的调优脚本，需要对以下几个对象进行调整：

- **optimizers**  
编译器的开/关二元选项列表，对于 GCC，可以使用 yaTuner 内置工具进行获取，并在此基础上手动进行移除或添加。
- **parameters**  
编译器的参数选项列表，对于 GCC，可以使用 yaTuner 内置工具进行获取。
- **comp 函数**  
接收调优器所给的选项，对程序进行编译。设计使用这一函数的目的在于适应不同的编译器并且能够为不同项目所使用的种类繁多的构建系统提供支持。
- **run 函数**  
通过执行或其它方式获取此程序的性能、体积等优化目标的数值，并反馈到调优器。
- **perf 函数**  
返回程序的特征用于调优。可以使用 yaTuner 内置的工具进行获取，也可以自定义程序特征的获取方式。
- 其它有关的调优时采样数、迭代次数等。

## （二） 多目标调优支持

在调优过程中，yaTuner 会通过 **run** 这一在调优脚本中定义的函数获取调优目标的结果。例如，对于程序运行速度（时间）的优化，一个简单的 **run** 的实现如下：

```

1 def run():
2     return yatuner.utils.fetch_perf_stat('./build/matmul.exe')['duration_time'] / 1000

```

而对于程序体积，可以实现如下：

```

1 def run():
2     return yatuner.utils.fetch_size('./build/matmul.exe')

```

对于 **run** 这一函数返回值的定义直接与调优目标相关，采用这种方式获取调优目标的结果可以在一定程度上兼容不同程序的特殊性，提高调优的可拓展性。

### (三) 自动获取编译器优化选项

由于编译器选项的繁多，处于实践考虑，yaTuner 支持了自动获取 GCC 当前版本的优化选项。

yaTuner 通过执行 `gcc --help=optimizers` 并通过正则表达式对返回结果进行处理获取支持的 Optimizers；通过执行 `gcc -Q --help=params` 并通过正则表达式对返回结果进行处理获取支持的 Parameters 范围以及默认值。

此外，可以通过修改优化脚本在自动获取的基础上手动添加要优化的 Optimizers 和 Parameters。对于非 GCC optimizers 的纯开/关二元编译选项（例如对于库链接与否的取舍），也可以添加到 Optimizers 这一列表中进行调优。

### (四) 执行优化

对于 Optimizers 这一类优化选项，yaTuner 首先使用假设检验的方法进行一次筛选，提取出可能对程序调优目标具有优化效果的优化选项。在此基础上使用 GPyOpt [2] 工具的贝叶斯优化器进行进一步组合和筛选，初步达到一个较优的结果。

继而，对于 Parameters 使用算法2进行筛选，在此基础上由用户在调优脚本中选择使用贝叶斯优化器或使用 yaTuner 实现的 LinUCB 优化器对参数进行调整以实现优化。

### (五) 生成报告

运行结束后 yaTuner 将在指定的工作区文件夹中自动生成一系列文件与图表，报告调优结果。

此处以 PolyBench 中的 deriche 程序为例，在使用 yaTuner 对其进行调优之后，可以在其工作目录中得到以下几个文件：

LinUCB（如果使用的话）的收敛过程：

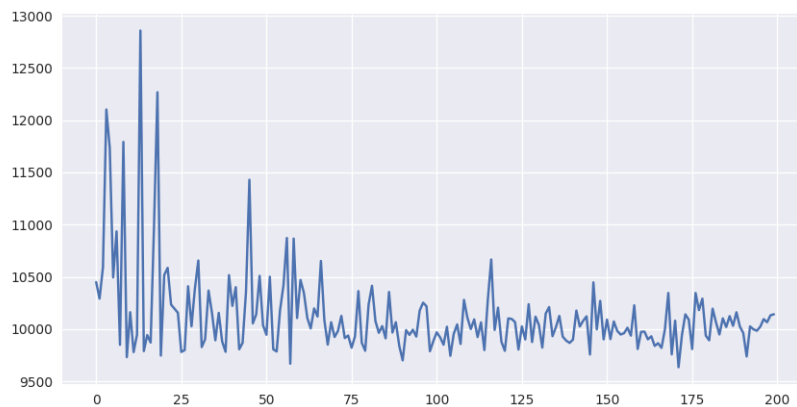


图 3: LinUCB 收敛过程

与其它优化选项相比较的测试结果小提琴图：

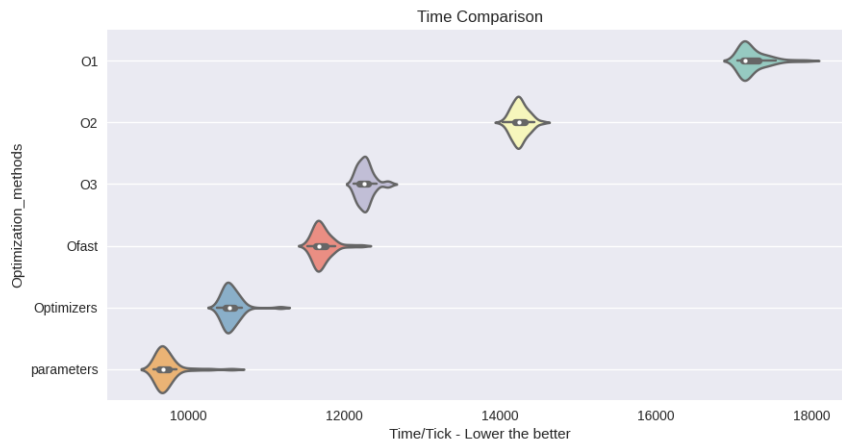


图 4: 测试结果小提琴图

筛选出的 Optimizers (位于 `selected_optimizers.txt`) :

```
1 -funroll-all-loops
2 -ffast-math
```

筛选出的 Parameters<sup>9</sup> (位于 `selected_parameters.txt`) :

```
1 align-loop-iterations
2 analyzer-max-enodes-per-program-point
3 asan-instrument-writes
4 asan-stack
5 comdat-sharing-probability
6 hot-bb-frequency-fraction
7 inline-heuristics-hint-percent
8 ipa-cp-max-recursive-depth
9 ipa-max-agg-items
10 lim-expensive
11 loop-block-tile-size
12 loop-invariant-max-bbs-in-loop
13 loop-max-datarefs-for-datadepts
14 max-average-unrolled-insns
15 max-combine-insns
16 max-completely-peeled-insns
17 max-cse-insns
18 max-debug-marker-count
19 max-delay-slot-live-search
20 max-find-base-term-values
21 max-fsm-thread-length
```

<sup>9</sup>由于篇幅限制有所省略。

```

22 max-grow-copy-bb-insns
23 max-inline-insns-single
24 max-inline-insns-size
25 max-partial-antic-length
26 ...

```

调优后得到的 Parameters<sup>10</sup> (位于 `optimized_parameters.txt`) :

```

1 align-loop-iterations 4
2 analyzer-max-enodes-per-program-point 38
3 asan-instrument-writes 0
4 asan-stack 0
5 comdat-sharing-probability 110
6 hot-bb-frequency-fraction 2413
7 inline-heuristics-hint-percent 689686
8 ipa-cp-max-recursive-depth 33
9 ipa-max-agg-items 49
10 lim-expensive 186
11 loop-block-tile-size 175
12 loop-invariant-max-bbs-in-loop 31034
13 loop-max-datarefs-for-datadepts 1724
14 max-average-unrolled-insns 165
15 max-combine-insns 2
16 max-completely-peeled-insns 827
17 max-cse-insns 6206
18 max-debug-marker-count 379310
19 max-delay-slot-live-search 2296
20 max-find-base-term-values 896
21 max-fsm-thread-length 689654
22 max-grow-copy-bb-insns 63
23 max-inline-insns-single 627
24 max-inline-insns-size 0
25 max-partial-antic-length 931
26 ...

```

除上述几个重要文件外，对预测试运行的结果汇总以及数据记录等文件也存储在工作区内。

---

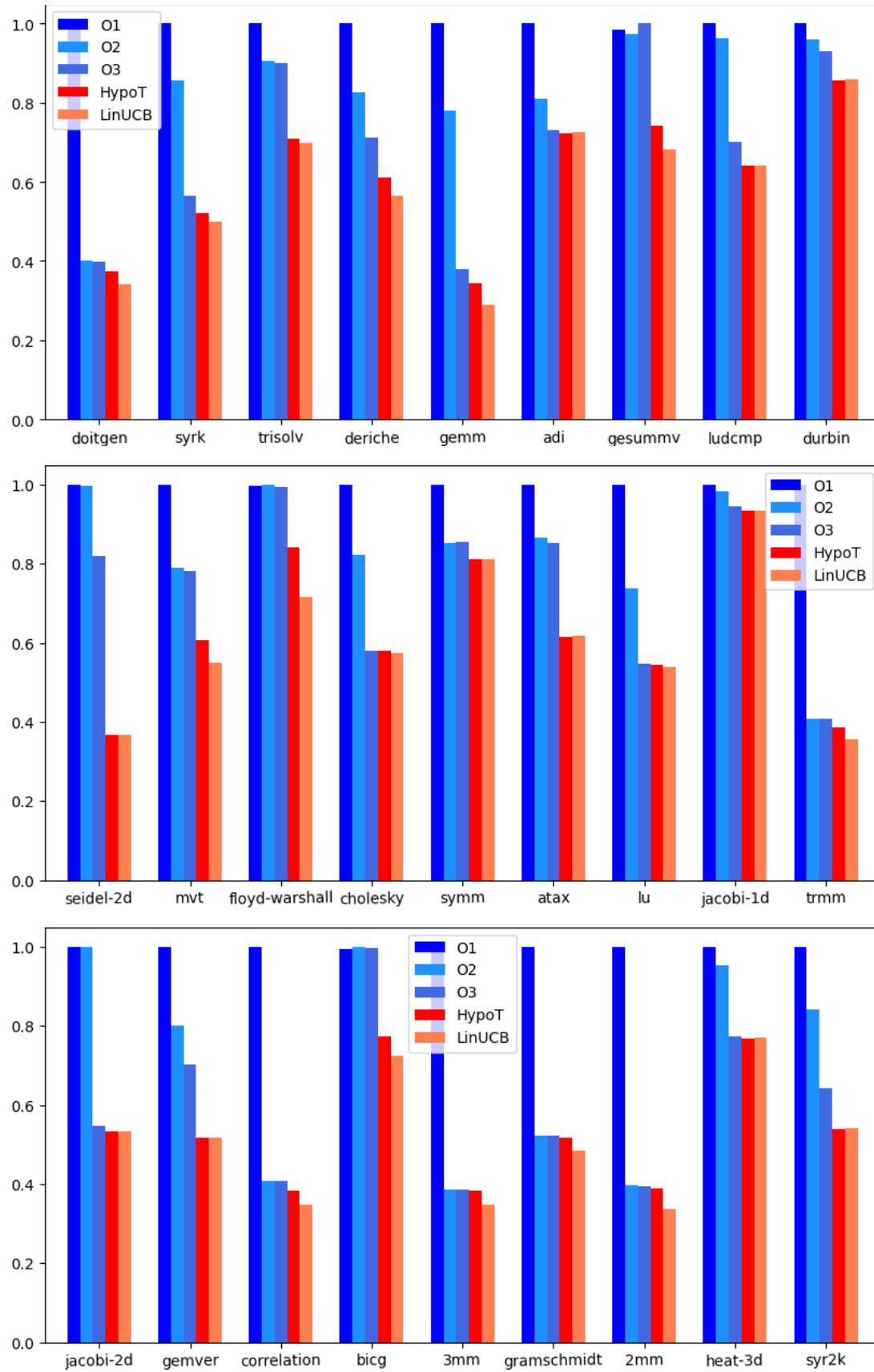
<sup>10</sup>由于篇幅限制有所省略。

## 六、 测试结果

### (一) PolyBench

Case Name	O1	O2	O3	HypoT <sup>11</sup>	LinUCB <sup>12</sup>	$\Delta_{HypoT}$ <sup>13</sup>	$\Delta_{LinUCB}$
doitgen	35.66	<b>14.31</b>	14.23	13.37	<b>12.14</b>	6.59%	15.15%
syrk	12.57	<b>10.77</b>	7.10	6.55	<b>6.28</b>	39.21%	41.66%
trisolv	2.25	<b>2.04</b>	2.03	1.60	<b>1.57</b>	21.62%	22.93%
deriche	17.23	<b>14.25</b>	12.26	10.54	<b>9.72</b>	25.99%	31.81%
gemm	25.77	<b>20.12</b>	9.79	8.83	<b>7.44</b>	56.12%	63.01%
adi	196.84	<b>159.34</b>	144.01	<b>142.45</b>	142.93	10.60%	10.30%
gesummv	2.46	<b>2.43</b>	2.50	1.85	<b>1.70</b>	23.93%	30.08%
ludcmp	235.73	<b>227.22</b>	165.48	151.27	<b>150.85</b>	33.42%	33.61%
durbin	0.96	<b>0.92</b>	0.89	<b>0.82</b>	0.82	10.63%	10.31%
fdtd-2d	78.93	<b>63.61</b>	40.68	<b>39.88</b>	39.92	37.31%	37.25%
seidel-2d	320.16	<b>318.99</b>	262.60	117.33	<b>117.32</b>	63.22%	63.22%
mvt	4.10	<b>3.24</b>	3.21	2.49	<b>2.26</b>	23.09%	30.24%
floyd-warshall	287.93	<b>288.67</b>	286.73	242.93	<b>206.84</b>	15.84%	28.35%
cholesky	248.58	<b>204.17</b>	144.02	144.36	<b>142.58</b>	29.29%	30.16%
symm	18.90	<b>16.13</b>	16.15	15.36	<b>15.31</b>	4.80%	5.08%
atax	3.46	<b>2.99</b>	2.95	<b>2.12</b>	2.14	29.01%	28.56%
lu	303.91	<b>224.68</b>	166.33	165.77	<b>164.11</b>	26.22%	26.96%
jacobi-1d	0.80	<b>0.79</b>	0.76	0.75	<b>0.75</b>	5.10%	5.11%
trmm	25.67	<b>10.46</b>	10.45	9.94	<b>9.11</b>	5.06%	12.91%
nussinov	66.90	<b>31.05</b>	31.12	25.28	<b>22.51</b>	18.59%	27.48%
jacobi-2d	67.37	<b>67.46</b>	36.93	36.08	<b>36.05</b>	46.51%	46.56%
gemver	4.96	<b>3.97</b>	3.49	2.56	<b>2.56</b>	35.52%	35.55%
correlation	42.12	<b>17.17</b>	17.20	16.21	<b>14.62</b>	5.57%	14.86%
bicg	3.47	<b>3.49</b>	3.49	2.70	<b>2.53</b>	22.51%	27.66%
3mm	122.62	<b>47.54</b>	47.50	46.92	<b>42.68</b>	1.30%	10.23%
gramschmidt	43.07	<b>22.52</b>	22.52	22.31	<b>20.86</b>	0.96%	7.36%
2mm	79.95	<b>31.82</b>	31.65	31.07	<b>27.07</b>	2.35%	14.95%
heat-3d	76.93	<b>73.32</b>	59.54	<b>59.03</b>	59.25	19.48%	19.18%
syr2k	28.02	<b>23.57</b>	18.00	<b>15.11</b>	15.16	35.92%	35.70%
covariance	41.86	<b>16.83</b>	16.86	15.79	<b>14.06</b>	6.17%	16.42%

表 3: PolyBench 调优结果汇总<sup>14 15</sup><sup>11</sup> 使用假设检验及贝叶斯优化筛选 Optimizers 之后的运行结果。<sup>12</sup> 在 HypoT 的基础上筛选 Parameters 并使用 LinUCB 优化之后的运行结果。<sup>13</sup> 性能提升百分比的计算方法:  $\Delta = \frac{P - P_{O2}}{P_{O2}}$ , 其中  $P$  为优化后的性能,  $P_{O2}$  为 gcc 编译器仅打开 O2 选项编译后的性能。<sup>14</sup> 除性能提升百分比外, 本表中其余数据单位为毫秒。<sup>15</sup> 蓝色标注为 O2 运行结果, 红色标注为此表所列选项中某程序的最优结果。

图 5: Polybench 结果统计<sup>16</sup>



## (二) 其它程序

Case Name	O1	O2	O3	HypoT	LinUCB	$\Delta_{HypoT}$	$\Delta_{LinUCB}$
Matmul	351.74	<b>289.04</b>	331.05	<b>65.14</b>	80.53	77.46%	72.13%
RayTracer	280.41	<b>242.75</b>	242.72	211.36	<b>197.17</b>	12.93%	18.77%
TSP GA	383.29	<b>393.20</b>	378.78	352.99	<b>346.06</b>	10.22%	11.99%
Triple <sup>17</sup>	976.68	<b>888.51</b>	914.41	605.19	<b>602.26</b>	31.88%	32.21%

表 4: 调优结果<sup>18</sup>

## 七、 总结

本项目通过使用假设检验、贝叶斯优化以及 LinUCB 等算法及工具实现了一个对编译进行自动调优的工具，实际测试表明其能够在编译优化上取得较为良好的效果。由于采用了假设检验的方式减少参数的搜索空间，yaTuner 相比于直接搜索可以获得更快的速度和更好的效果以及鲁棒性。其次，自动生成调优脚本辅之以手动微调的设计有利于实践过程中适应不同的调优目标和内容。

除此以外，通过手动编辑 yaTuner 的调优脚本，还可以将其应用到其它与选项开关、参数选择有关的超参数优化的调优场景中，并支持多目标（如降低缓存占用、提高分支预测命中率等场景）的优化，在实际项目优化中具有较高的应用价值。

## Acknowledgement

本项目得到了来自华为的服务器环境支持，特别感谢林达老师在比赛过程中提供的帮助以及辛苦付出。

在项目进展过程中：

郭大玮同学针对前期调研中发现的问题创新性地提出了运用假设检验方法对优化器及参数进行降维，并完成了 LinUCB 及假设检验在本应用场景中重要的可行性验证，在工具形成的过程中优化了调优结果的图表及格式。

梅骏逸同学在前期调研中复现了前人的工作并尝试进行了其它方法的可行性验证，完成了项目框架与测试平台的搭建，并完成了后期大部分代码的重构、封装以及文档的书写工作，承担了程序的工具设计和打包封装工作。

<sup>16</sup>表中为数据为归一化处理后的运行时间，对于某一个程序，特定优化方法在图中的数据由其运行时间除以同程序下的最长运行时间。对于同一个程序，图中数据越小代表运行时间越短，即结果越好。

<sup>17</sup>结合了 Matmul, RayTracer, TSP-GA 三个程序，这三个程序均来自于 Opentuner 对 g++ 的调优示例。

<sup>18</sup>此表详细说明与表3一致。

## 参考文献

- [1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, Aug 2014.
- [2] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. <http://github.com/SheffieldML/GPyOpt>, 2016.
- [3] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*, 2022.
- [4] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web - WWW '10*. ACM Press, 2010.