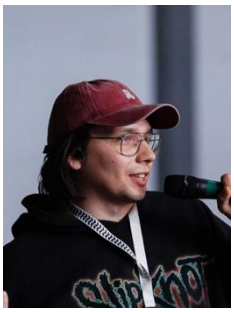
DOI https://doi.org/10.46845/2541-8254-2025-2(49)-26-26
УДК 004.4'23



ОПТИМИЗАЦИЯ ПРОЦЕССОВ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ МНОГОПОЛЬЗОВАТЕЛЬСКИХ СИСТЕМ С ПОМОЩЬЮ ЧАСТИЧНОГО ИЛИ ПОЛНОГО ПЕРЕХОДА НА КОНКУРЕНТНЫЕ ВЫЧИСЛЕНИЯ

А. В. Некрасов, студент
E-mail: poksliun@gmail.com
ФГАОУ ВО «Балтийский федеральный университет имени Иммануила Канта»

С. Д. Верещагин, канд. физ.-мат. наук, доцент
E-mail: sergey.ver@gmail.com
ФГАОУ ВО «Балтийский федеральный университет имени Иммануила Канта»

В статье рассмотрен вопрос о целесообразности многопроцессорных вычислений, основанных на принципах вытесняющей многозадачности с помощью инструмента pytest-xdist в тестировании многопользовательских систем с использованием фреймворка для тестирования pytest. Приведен пример более оптимального подхода, основанного на комбинации принципов кооперативной и вытесняющей многозадачностей, реализованного в инструменте собственной разработки: pytest-and. Проведен сравнительных анализ предложенного решения с существующими инструментами для оптимизации процессов автоматизированного тестирования.

Ключевые слова: *ООП, асинхронное программирование, многозадачность, автоматизированное тестирование, Python.*

ВВЕДЕНИЕ

Высокопроизводительные многопользовательские системы все чаще используют асинхронные средства языков программирования при работе с операциями ввода-вывода. Использование неблокирующих операций ввода-вывода позволяет более эффективно использовать ресурсы вычислительных машин, а также перераспределять нагрузку на другие сервисы, реализуя так называемый тонкий клиент на стороне, отправляющей запросы на более мощное серверное решение (или толстый клиент). Главной проблемой использования асинхронных инструментов является невозможность эффективного комбинирования процессорозависимые вычислений и операций ввода-вывода в рамках одного запущенного процесса. Скрипты для автоматизированного тестирования на уровне модульного или интеграционного тестирования чаще всего либо взаимодействуют с тестируемым сервисом через тонкий клиент, либо сами являются таковым, из чего следует, что все сложные, процессорозависимые вычисления проводятся на стороне сервиса, а тестирующая сторона лишь «следит» за корректностью вычислений и изменением состояния сервиса.

Сообщество python-разработчиков создало некоторое количество инструментов для оптимизации автоматизированных тестов, которые работают как по принципам вытесняющей многозадачности, так и по принципам кооперативной многозадачности [1], но не одновременно с двумя. Уклон в использование лишь одного принципа несет за собой как ограничения, не позволяющие использовать все возможные ресурсы вычислительных систем, так и ограничения, из-за которых увеличивается время выполнения автоматизированных сценариев.

ОБЪЕКТ ИССЛЕДОВАНИЯ

Объектом исследования является разработанный инструмент (плагин) для создания как асинхронных, так и синхронных автоматизированных тестов на основе фреймворка pytest.

Как и основной инструмент ускорения тестовых прогонов – pytest-xdist, разработанный инструмент основан на распределении предоставленных задач на некоторое количество процессов [2]. Ключевым отличием является наличие трех разных реализаций процессов, оптимизированных под выполнение различных типов задач. В предложенной реализации выделяется три типа задач: счетные задачи, блокирующие задачи ввода-вывода и неблокирующие задачи ввода-вывода. При запуске тестового прогона подразумевается, что каждая задача передается соответствующему ей процессу, для этого используется механизм маркирования тестов [3], который помогает точно определить тип задачи.

ЦЕЛИ И ЗАДАЧИ ИССЛЕДОВАНИЯ

Целью исследование является подтверждение или опровержение предположения об оптимизации процесса автоматизированного тестирования благодаря совместному использованию подходов кооперативной и вытесняющей многозадачностей в связке с фреймворком для автоматизированного тестирования на языке программирования Python: pytest.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Подготовить наиболее распространенные сценарии в тестировании многопользовательских систем.
2. Провести тестовые запуски на различных конфигурациях инструментов оптимизации.
3. Провести сравнение полученных результатов и выявить более эффективный подход к автоматизации тестирования многопользовательских систем.

МЕТОДЫ ИССЛЕДОВАНИЯ

В ходе работы использованы такие методы, как описание, сравнение, анализ и моделирование.

РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ

Для проведения исследования был подготовлен набор из трех возможных видов нагрузки в виде тестовых сценариев, а именно: тесты, содержащие счетный код (рекурсивный поиск 35 числа ряда Фибоначчи без использования кеширования) в количестве ста штук; веб-тесты, использующие блокирующую библиотеку requests для работы с сетью в количестве двухсот штук; веб-тесты, использующие неблокирующую библиотеку aiohttp для работы с сетью в количестве одной тысячи штук. Таким образом, в общей сложности будет 1 300 тестов.

Для возможности проверки производительности в близких к реальным условиям для веб-тестов было разработано приложение, способное принимать два типа http-запросов: post и get. Для большего соответствия реальным условиям в методах серверного приложения были добавлены задержки в 50 с, которые симулируют реальные долгие вычисления, проводимые на серверах в многопользовательских приложениях. Веб-приложение было создано на основе веб-фреймворка fastapi из-за его возможности использовать асинхронные интерфейсы и соответственно высокой скорости работы.

Для отслеживания использования ресурсов системы (загруженность CPU и использованная память RAM) была использована утилита sar (System Activity Report), снимающая снапшот системы каждую секунду.

Все тестовые запуски проводились на ThinkPad L14 gen 5 AMD Ryzen 7 PRO 7735U, 32 DDR5 RAM, 512 ROM. Из-за особенностей работы с многоядерными системами в Python программа запускалась внутри подсистемы WSL второй версии. Тестовое серверное приложение было развернуто на MacBook Pro m1 16 RAM, 512 ROM.

Тестовые запуски с использованием инструмента pytest-xdist проводились в двух конфигурациях:

- автоматическая конфигурация на 8 процессов (по количеству физических ядер);
- распределение тестов на 16 процессов (по количеству логических ядер);
- распределение тестов на 100 процессов (подобранное вручную значение).

Стандартные верхнеуровневые интерфейсы для создания многопроцессорных приложений при автоматическом определении необходимого числа создаваемых процессов опираются на количество процессорных ядер вычислительной машины, для получения достаточного ускорения при нетронутой стабильности системы, как, например, это было сделано в модуле ProcessPoolExecutor стандартной библиотеки Python [4] или в автоматической конфигурации инструмента pytest-xdist. Вопреки автоматическому подбору количества создаваемых процессов последняя конфигурация подразумевает создание ста процессов. Такое решение было принято в связи с тем, что тестовая выборка в большей степени состоит из задач ввода-вывода, из-за чего операция вытеснения задач не является особенно критичной. Такое количество создаваемых процессов было подобрано эмпирическим путем и в первую очередь было необходимо для ускорения инструмента pytest-xdist, дальнейшее добавление процессов сверх ста приводило к просадкам производительности.

Тестовые запуски с использованием инструмента собственной разработки также проводились в двух конфигурациях:

- 5 процессов, 200 потоков, 1 000 асинхронных задач;
- 100 процессов, 200 потоков, 1 000 асинхронных задач.

По результатам тестовых прогонов была составлена сравнительная таблица со следующими показателями:

- time – общее время, затраченное на выполнение всех тестов;
- user – процент времени, затраченного процессором на выполнение кода;
- idle – процент времени, когда процессор простаивает и не выполняет никаких задач;
- mem – использованная память в килобайтах.

Результаты тестовых запусков (таблица), показывают неэффективность использования только принципов вытесняющей многозадачности, как при распределении на 8 рекомендуемых процессах, так и на 100 подобранных при работе с доминирующим количеством задач ввода-вывода. Стоит отметить ускорение более чем в 10 раз при значительном увеличении количества созданных процессов от автоматического подбора, но данное решение несет за собой четырехкратное увеличение потребления оперативной памяти, что может привести к отказу вычислительной машины при выполнении реальных более комплексных тестовых сценариев. Также как и ожидалось, использование процессора можно считать незначительным при обеих конфигурациях тестовых запусков с использованием pytest-xdist.

Таблица – Результаты тестовых запусков с использованием двух инструментов в разных конфигурациях

Конфигурация	Метрики			
	time	user	idle	mem
pytest-xdist 8 processes	7324 с.	0.1977%	99.75%	974706 KB
pytest-xdist 100 processes	646 с.	9.7901%	89.90%	4343563 KB
pytest-and 5 processes 100 threads 1000 tasks	63 с.	77.754%	20.68%	3568443 KB
pytest-and 100 processes 100 threads 1000 tasks	68 с.	77.1416%	21.88%	3679011 KB

При рассмотрении использования ядер процессора и оперативной памяти на временной шкале (рисунок 1) можно увидеть подтверждение вышесказанному. При большом количестве созданных процессов система взаимодействовала в большей степени с ядрами процессора только первые 60 с, когда происходило выполнение ста счетных задач. В остальное время процессор почти не задействовался, и операции вытеснения не могли привести к заметным артефактам.

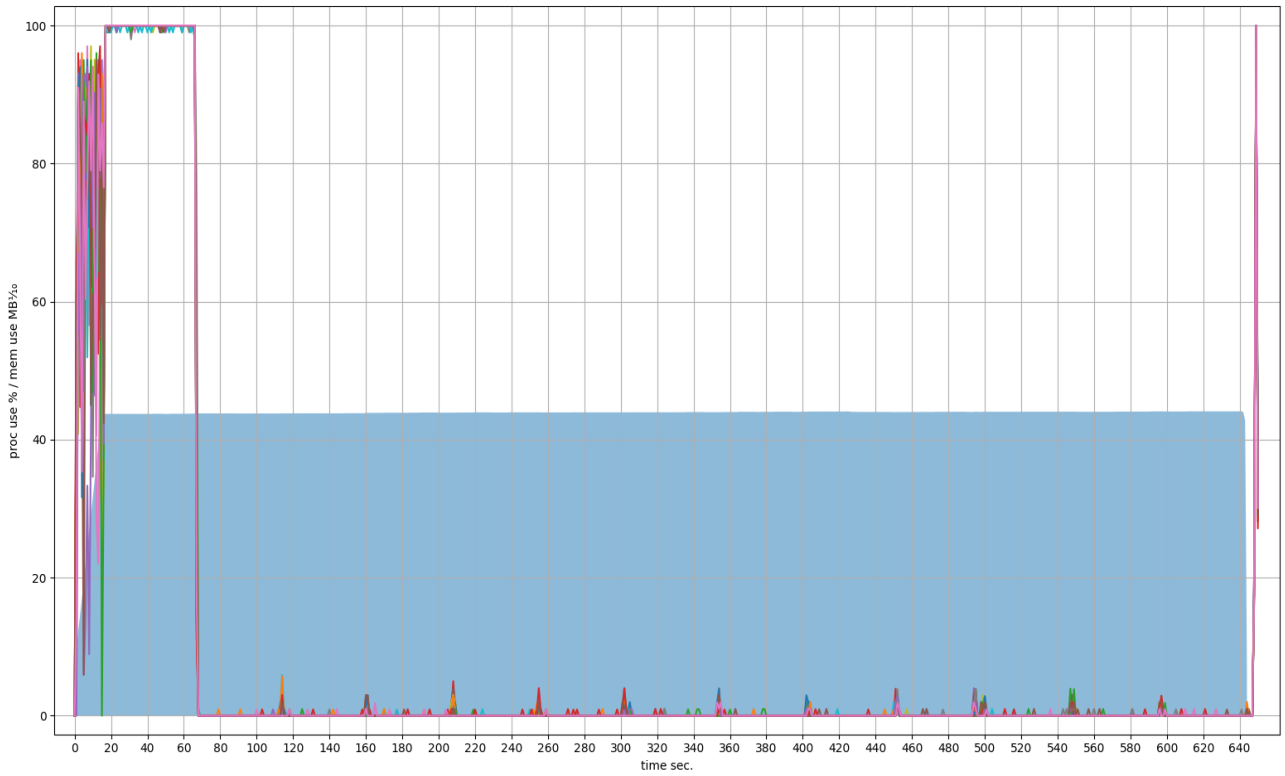


Рисунок 1 – График использования ядер процессора и оперативной памяти при использовании инструмента pytest-xdist с конфигурацией 100 процессов

В свою очередь, комбинированный подход, представленный в pytest-and, принес стократное ускорение относительно первой конфигурации pytest-xdist и десятикратное ускорение относительно второй соответственно (таблица). Стоит отметить, что выделенное количество процессов под счетные задачи во второй конфигурации тестового запуска с помощью pytest-and не принесло дополнительного ускорения, как это было с pytest-xdist, а, наоборот, замедлило тестовый прогон. В этом случае операция вытеснения оказывает сильное влияние на эффективность инструмента, так как большую часть времени он использует вычислительные мощности процессора. Данный факт можно увидеть на временном графике использования ядер процессора и памяти (рисунок 2).

При анализе графика (рисунок 2) была предложена гипотеза, что текущая реализация инструмента неэффективно производит постобработку уже пройденных тестовых сценариев в процессе, выполняющем асинхронные задачи, вследствие чего средняя загруженность процессора составила 77 %.

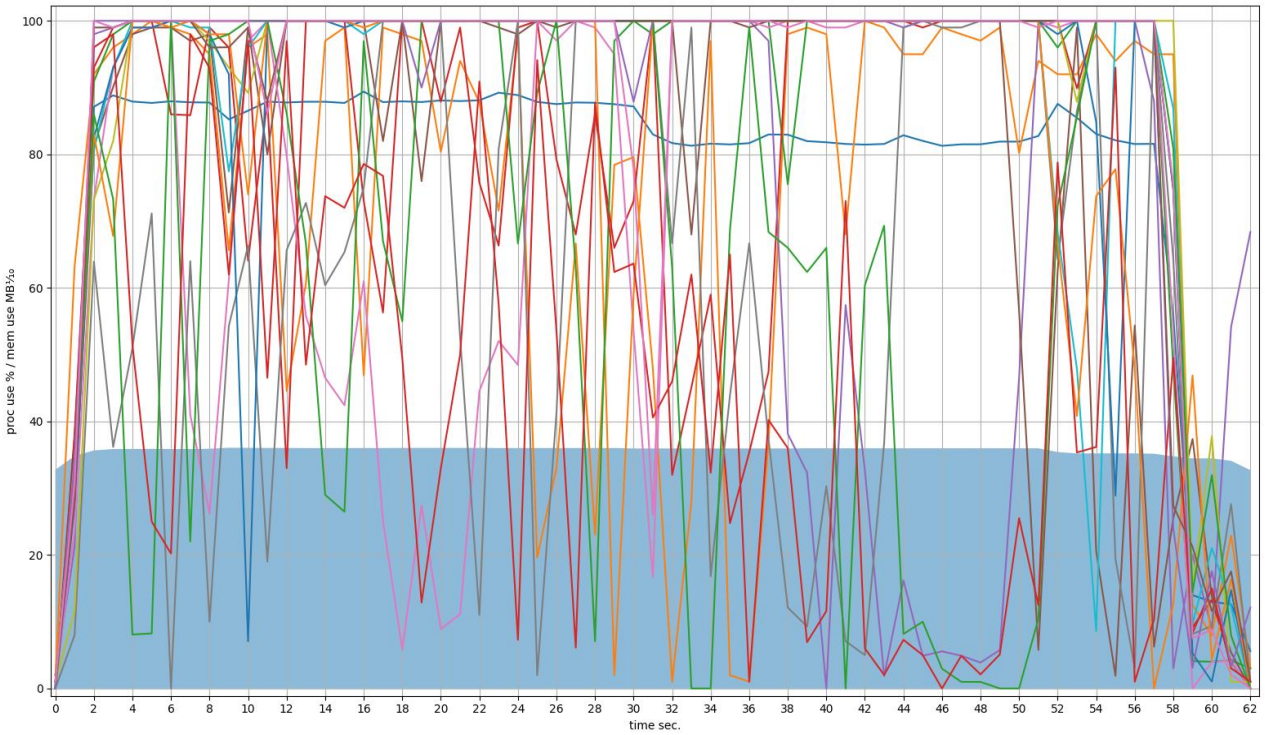


Рисунок 2 – График использования ядер процессора и оперативной памяти при использовании инструмента pytest-and с конфигурацией 5 процессов, 100 потоков, 1 000 задач

При этом загруженность процессора в 77 % можно рассматривать как запас производительности в счетных задачах и в задачах ввода-вывода, так как при сборе метрик было выявлено среднее время выполнения ста счетных задач в течение 45 с. Похожую закономерность видно и на временном графике (рисунок 2): после 42-й секунды три процесса приложения использовали не более 16 % процессорного времени.

ЗАКЛЮЧЕНИЕ

Таким образом, использование комбинации принципов кооперативной и вытесняющей многозадачностей в тестировании многопользовательских систем позволило повысить эффективность автоматизированного тестирования за счет уменьшения времени выполнения тестовых прогонов и более оптимального использования вычислительных ресурсов. Несмотря на полученные результаты, исследуемый подход обладает достаточным потенциалом для большей оптимизации, а также для добавления нового функционала, позволяющего проводить более гибкое автоматизированное тестирование.

СПИСОК ЛИТЕРАТУРЫ

1. Фаулер, М. Asyncio и конкурентное программирование на Python / М. Фаулер. – Москва: ДМК, 2023. – 398 с.

2. Pytest-xdist [Электронный ресурс]. – URL: https://pytest-xdist.readthedocs.io/en/stable/ (дата обращения: 03.05.2025).

3. Working with custom markers [Электронный ресурс]. – URL: https://docs.pytest.org/en/stable/example/markers.html#marking-test-functions-and-selecting-them-for-a-run (дата обращения: 02.05.2025).

4. Concurrent.futures – Launching parallel tasks [Электронный ресурс]. – URL: https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ProcessPoolExecutor (дата обращения: 03.05.2025).

OPTIMIZATION OF AUTOMATED TESTING PROCESSES FOR MULTI-USER SYSTEMS BY PARTIALLY OR COMPLETELY SWITCHING TO CONCURRENT COMPUTING

A. V. Nekrasov, student
E-mail: poksliun@gmail.com
Immanuel Kant Baltic Federal University

S. D. Vereshchagin, Associate Professor
E-mail: sergey.ver@gmail.com
Immanuel Kant Baltic Federal University

This article addresses the feasibility of multiprocessor computing based on the principles of preemptive multitasking using the pytest-xdist tool in testing multi-user systems using the pytest testing framework. An example of a more optimal approach based on a combination of cooperative and preemptive multitasking principles implemented in a tool of our own development: pytest-and. A comparative analysis has been carried out of the proposed solution with the existing tools for optimizing automated testing processes.

Keywords: *OOP, asynchrony, multitasking, test automation, Python.*