Московский физико-технический институт (государственный университет)
Физтех-школа радиотехники и компьютерных технологий
Кафедра информатики и вычислительной техники

Использование мета-данных режима безопасных вычислений архитектуры «Эльбрус» для контроля корректности аргументов библиотечных процедур

Выпускная квалификационная работа
(магистерская диссертация)

студент: Ситников П. В.
научный руководитель: Жмурин А. В.
консультант: Федоров А. В.

Москва, 2023

Введение
Особенности РБВ

- Мета-данные это теги, определяющие тип данных в архитектуре Эльбрус
- Обращение к памяти только через дескриптор
- Аппаратное прерывание
  - При выходе за границы массива
  - Обращение к памяти не через дескриптор
  - Работа с неинициализированными данными

Дескриптор

База - 64 бита	
Размер 32 бита	Смещение 32 бита

Указатель

Указатель - 64 бита

Введение

Проблемы совместимости по исходному коду

- Дескриптор и указатель несовместимы
- Портирование кода трудоемко
  - Статический анализ компилятора не покрывает ошибки, возникающие при динамическом формировании структур
  - Внутри библиотеки возникает прерывание, если аргументом библиотечной функции подана неправильная структура

Дескриптор

<table>
<tr><td colspan="2">База - 64 бита</td></tr>
<tr><td>Размер
32 бита</td><td>Смещение
32 бита</td></tr>
</table>

Указатель

Указатель - 64 бита

Введение

- В системных библиотеках контроль аргументов отсутствует
- Программист подает в процедуру системной библиотеки некорректные аргументы
- Программа падает внутри системной библиотеки
- Причина возникновения прерывания неизвестна
- Компания затрачивает время на объяснение пользователю, что аргументы некорректны
- Требуется контроль аргументов на раннем этапе и выдача диагностических предупреждений

Цель работы

Разработать метод контроля корректности аргументов библиотечных процедур с использованием мета-данных режима безопасных вычислений архитектуры «Эльбрус».

Задачи

- Разработать принцип контроля аргументов
- Разработать способы динамической отладки кода в РБВ
    - Инструментацией кода приложения путем добавления заголовочного файла отладочной библиотеки
    - Динамической линковкой отладочной библиотеки к приложению
- Выполнить оценку деградации производительности
- Разработать генератор проверочных функций

Проверочный код

Алгоритм контроля аргументов

- Контроль аргументов с помощью метаданных (тегов)
- Для каждой функции строится функция контроля корректности аргументов
- При работе приложения подменяется запуск библиотечной процедуры на проверочную
- После контроля аргументов вызов процедуры библиотеки
- В случае выявления несоответствий выдается предупреждение с указанием места ошибки

Проверочный код

Базовые проверки

- Аргумент-дескриптор должен быть дескриптором
- Аргумент-функция должен быть функцией
- Контроль достаточности размера дескриптора
- Если известна структура аргумента-дескриптора, проверка дескриптора на заданный тип структуры

Проверочный код

Пример

void *memcpy(void *dst, const void * src, size_t len)

- Dst и src должны быть дескрипторами
- Src должен быть проинициализирован
- Размер дескриптора dst должен быть >= len
- Размер дескриптора src должен быть >= len

size_t strlen(const char *string)

- String должен быть дескриптором
- String должен быть проинициализирован
- String должен содержать нулевой байт

Метод отладки приложения

Инструментация кода

- Отладочная библиотека с набором функций контроля аргументов, компилируемая вместе с отлаживаемой программой
- Программист добавляет include в свой код.

+ Точная локализация ошибки
+ Возможность инструментации отдельных файлов
- Необходима пересборка отладочного кода программы
- Инструментация каждого файла

Метод отладки приложения
Динамическая линковка библиотеки

- Отладочная библиотека с набором функций контроля аргументов, линкуемая к отлаживаемой программе
- + Не требует пересборки кода для работы в отладочном режиме. Перезапуск осуществляется изменением переменной среды LD_PRELOAD
- - Затруднена локализация ошибки, если код оптимизирован

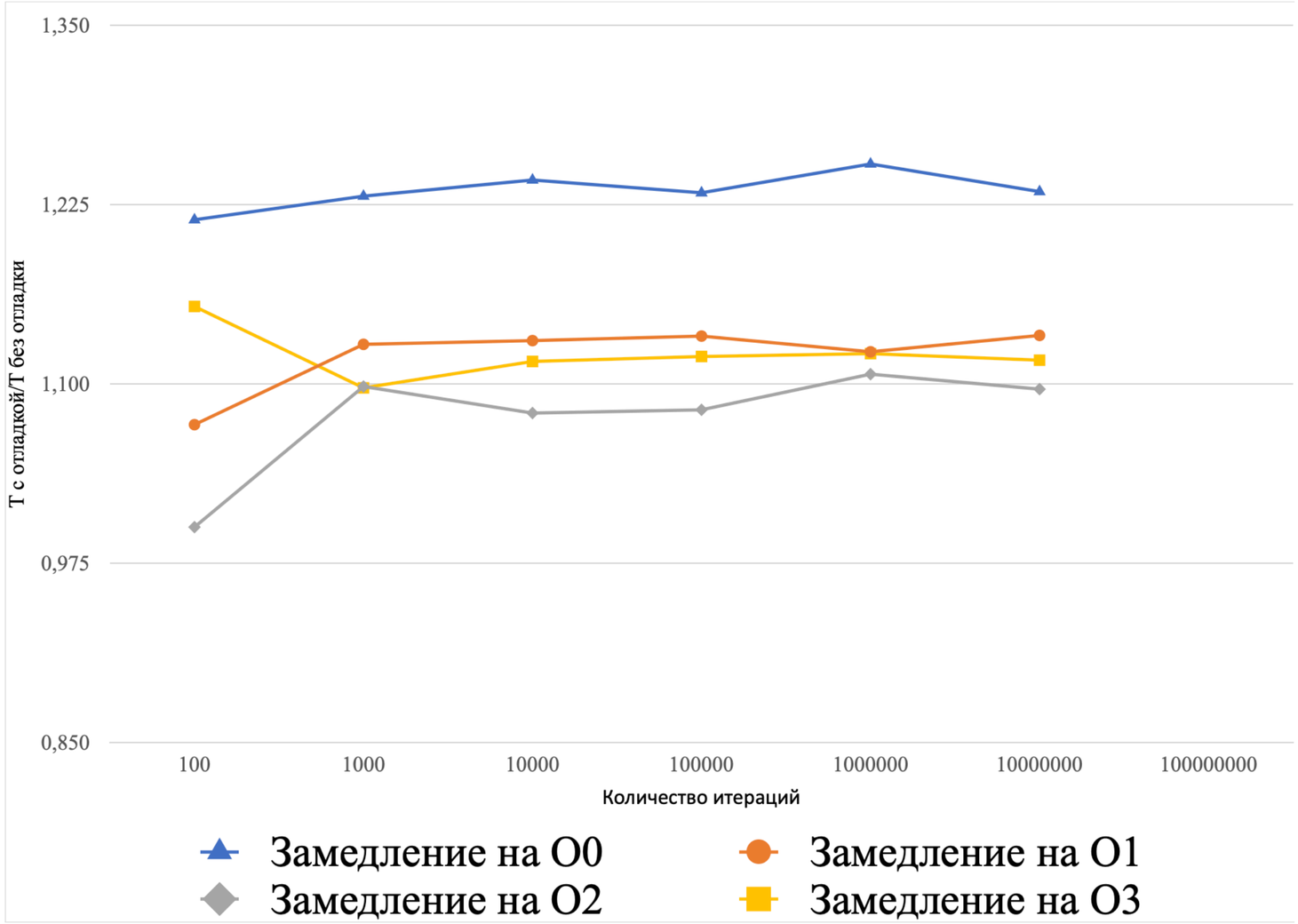
Оценка деградации производительности

- Синтетический стресс-тест
- Алгоритм
  - Сохранить время начала выполнения кода
  - Вызвать несколько библиотечных функций N раз N = 100,1000,10000,100000,1000000
  - Сохранить время окончания выполнения кода
  - Вычислить время выполнения цикла
- Замеры производились на функциях, активно работающих с памятью: strlen, memcpy, memcmp
- Верхняя оценка времени

Оценка деградации производительности

- 25% для O0
- 13% для O1
- 10% для O2
- 12% для O3

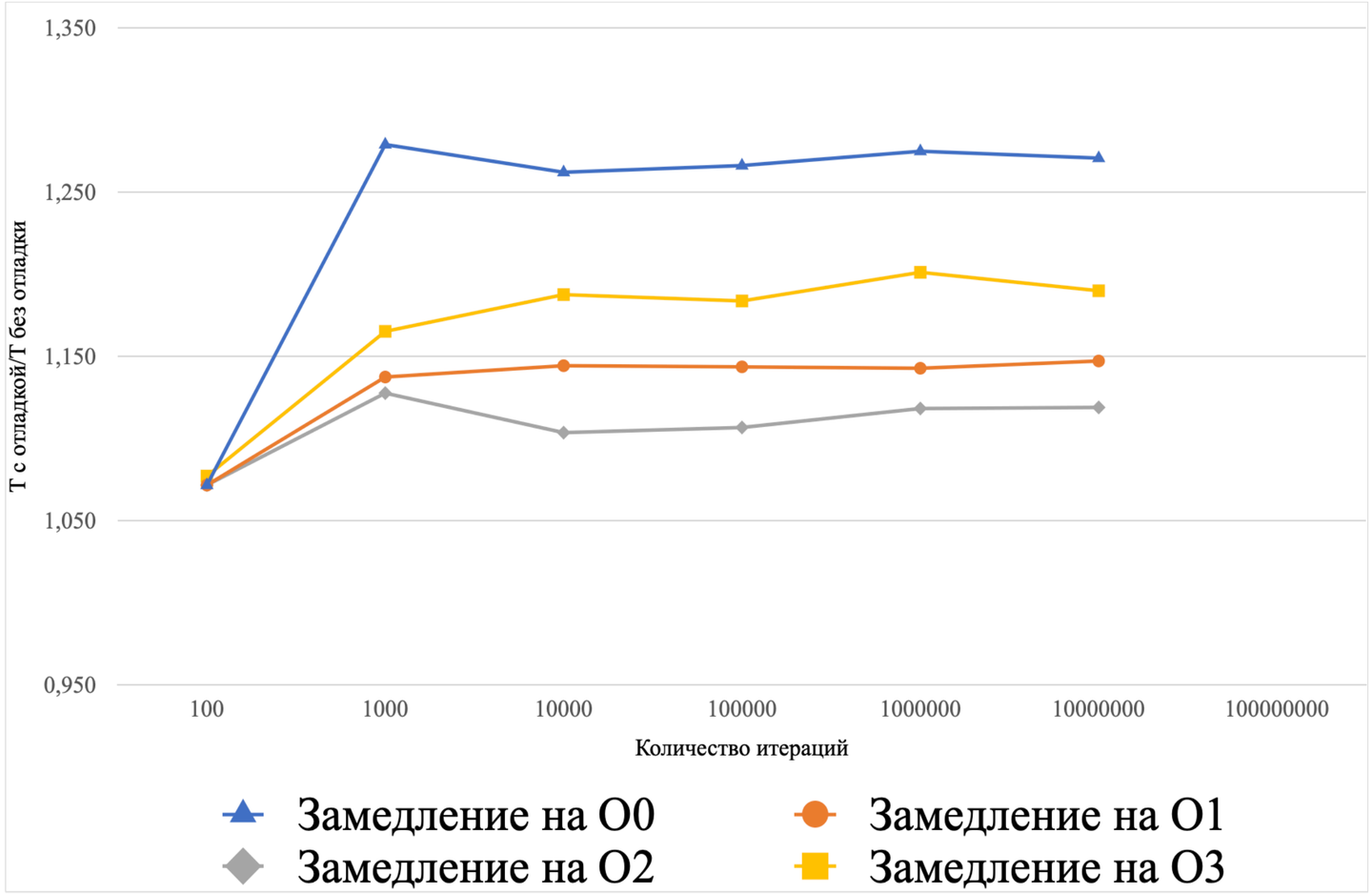
Инструментация кода



Оценка деградации производительности

Динамическая линковка

- 28% для O0
- 15% для O1
- 12% для O2
- 20% для O3



Генератор проверочных функций

- Большое количество процедур и библиотек
  - Glibc содержит более 200 процедур
- Простая структура базового кода контроля корректности аргументов
  - Создание заготовки отладочной библиотеки можно автоматизировать
  - Кардинально снижает затраты на разработку отладочных библиотек и снижает ошибки

Генератор проверочных функций

Генератор

- Реализован на языке python
- Задание
  - Описание прототипа функций в формате yaml
  - Заготовка отладочной библиотеки
- Результат – отладочная библиотека на языке C с проверочными функциями
- Применим для режима инструментации кода и для режима динамической линковки библиотеки

Генератор проверочных функций

Задание генератора

```
memcpy:
  ret_type: void*
  args:
    - { type: void*, name: dst }
    - { type: const void*, name: src }
    - { type: size_t, name: len }
  argchecks:
    - { check: descriptor, arg: [ dst, src ] }
    - { check: descriptor_size, arg: ap_size(src) >= len }
    - { check: descriptor_size, arg: ap_size(dst) >= len }
```

Сгенерированный фрагмент

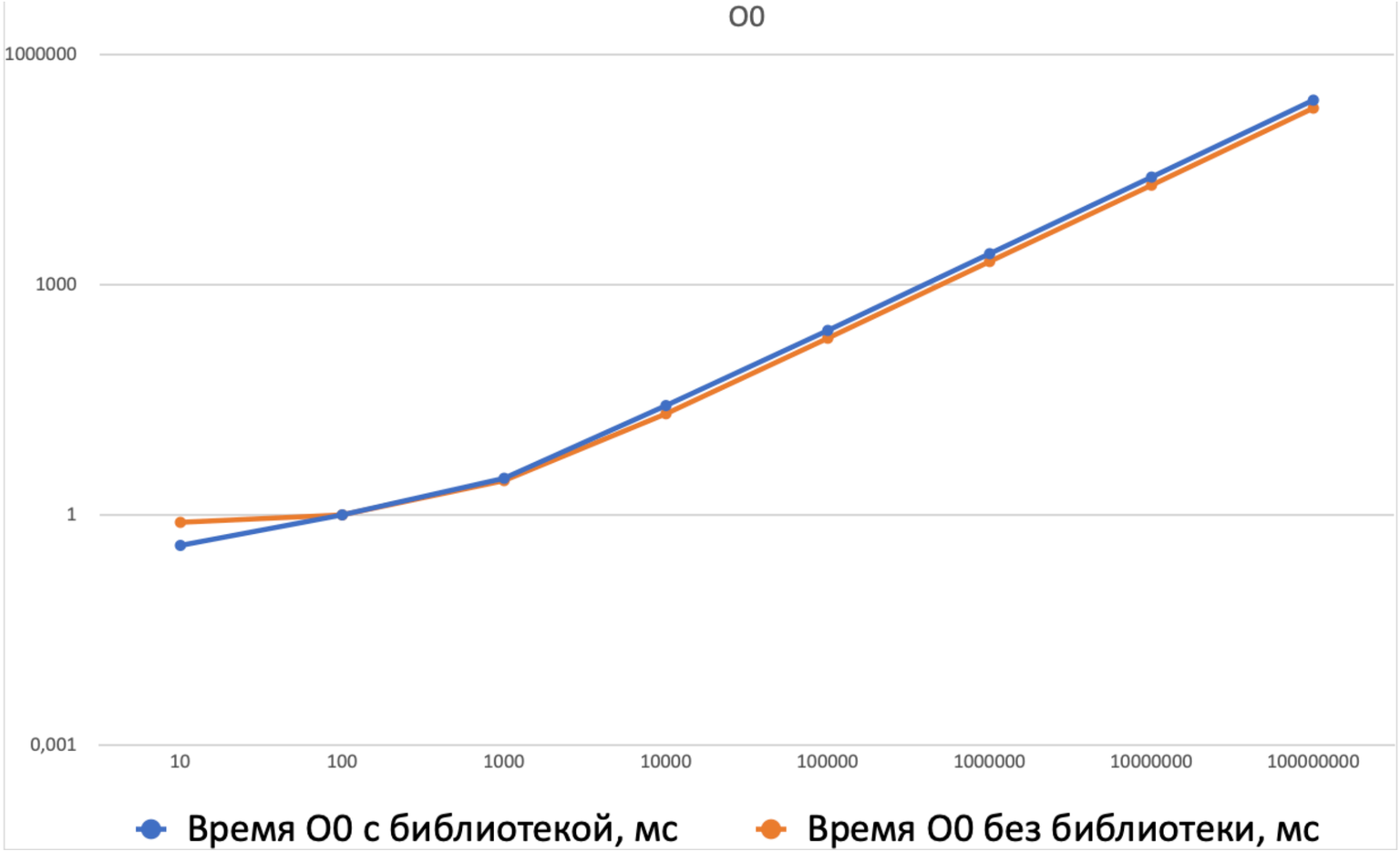
```
void *e2k_pm_memcpy(char * e2k_file, int e2k_line, const char * e2k_func , void* dst, const void* src, size_t len){

    //проверки
    int is_descriptor_dst = check_for_descriptor(e2k_file, e2k_line, e2k_func, 1, s1);
    if (!is_descriptor_dst) {
        return NULL;
    }

    int is_descriptor_src = check_for_descriptor(e2k_file, e2k_line, e2k_func, 2, s2);
    if (!is_descriptor_src) {
        return NULL;
    }

    if (!(e2k_get_ap_size(dst) >= len)) {
        e2k_ptr128_print_error(e2k_file, e2k_line, e2k_func, 1, ARG_IS_LESS_THAN_MAXLEN);
        return NULL;
    }

    if (!(e2k_get_ap_size(src) >= len)) {
        e2k_ptr128_print_error(e2k_file, e2k_line, e2k_func, 2, ARG_IS_LESS_THAN_MAXLEN);
        return NULL;
    }

// Вызов оригинальной функции
#ifndef __PIC__
    return memcpy(s1, s2, len);
#else
    return libc_memcpy(s1, s2, len);
#endif

}
```

Результаты

- Разработан принцип контроля аргументов
- Прототипирован метод отладки защищенного кода инструментацией кода приложения.
- Прототипирован метод отладки защищенного кода линковкой отладочной библиотеки
- Выполнена оценка деградации производительности
- Реализован генератор проверочных функций для базовых проверок

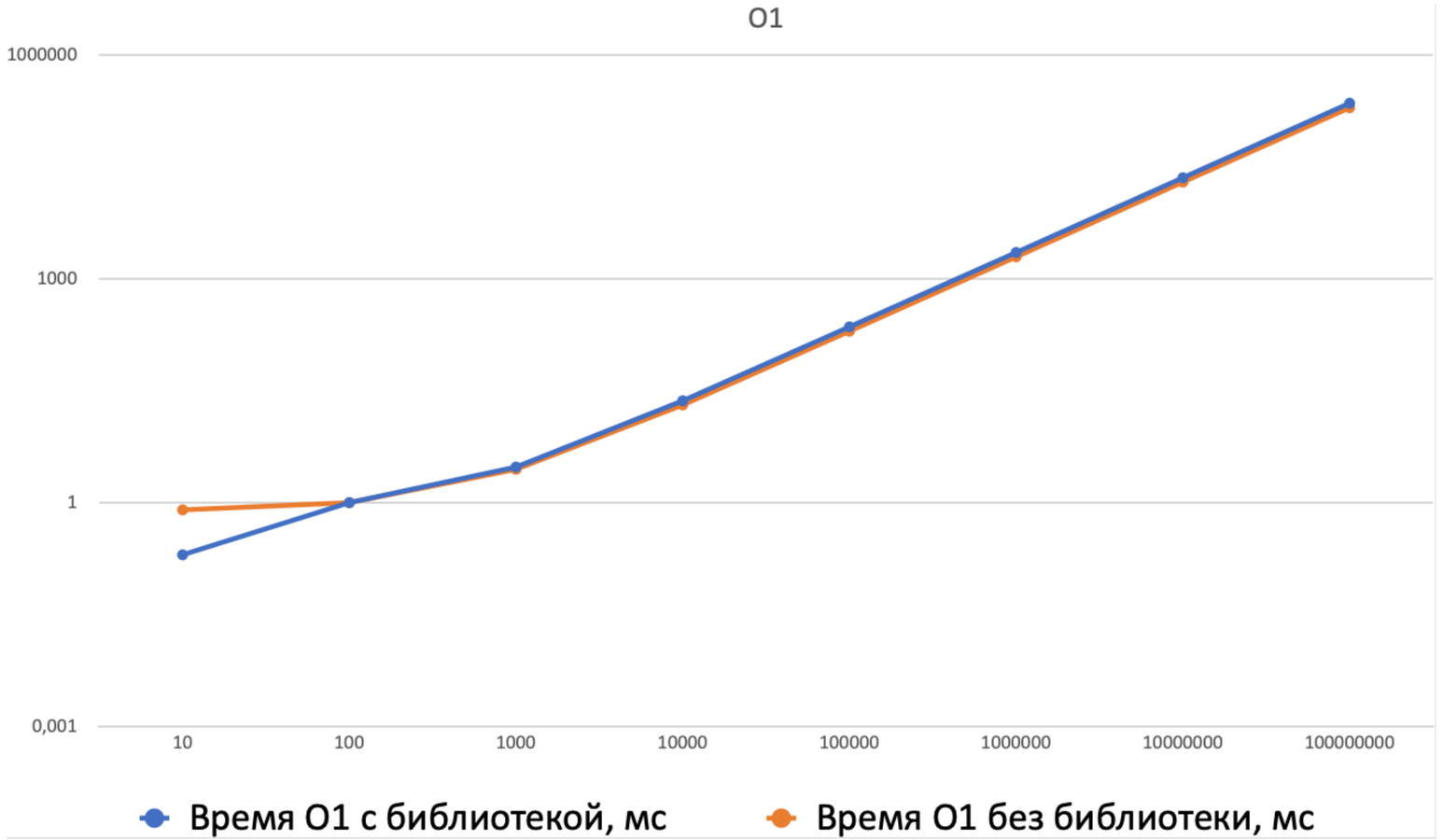
Время работы программы

O0



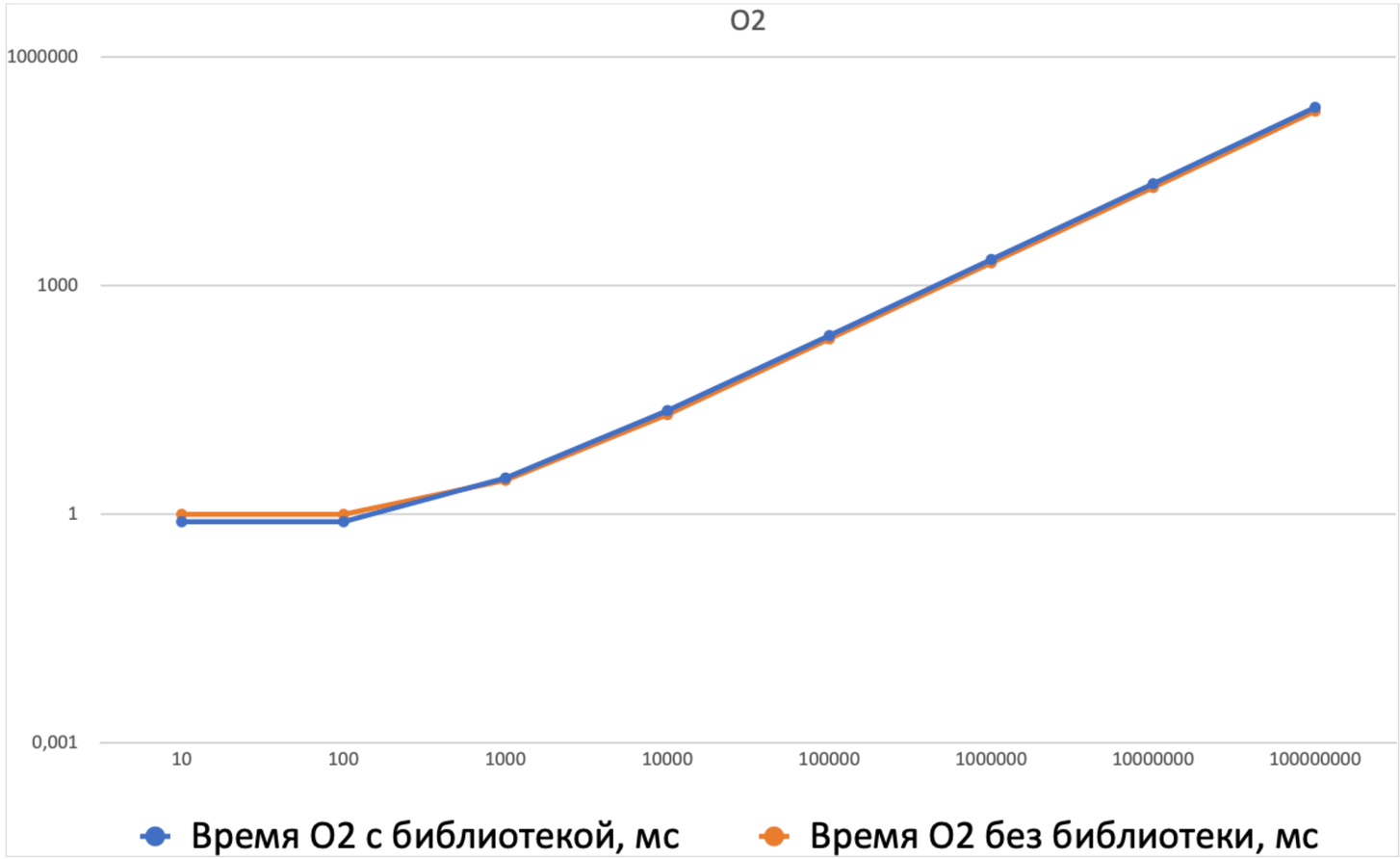
Время работы программы

O1



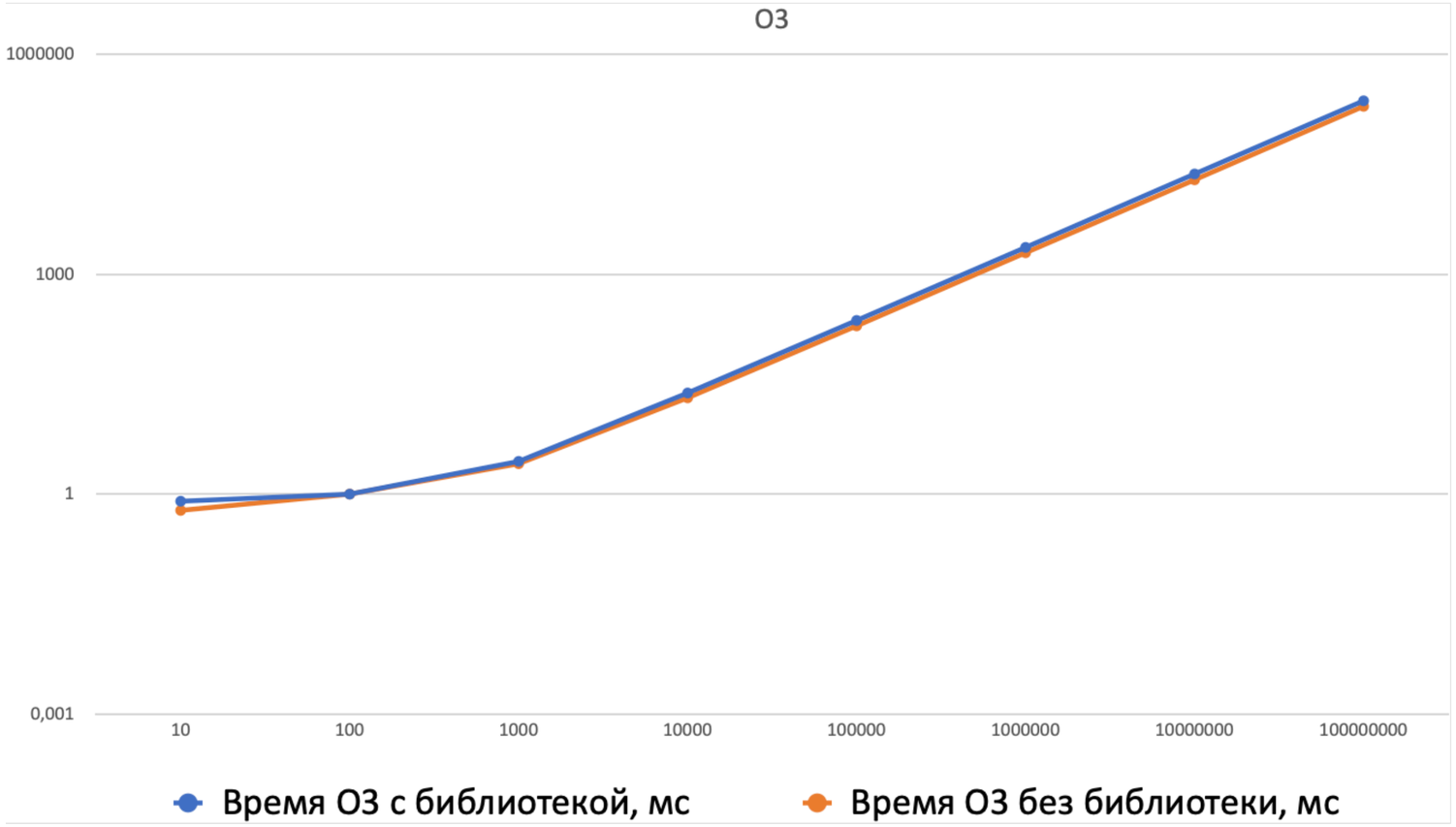
Время работы программы

O2



Время работы программы

O3



Оценка деградации производительности

Алгоритм

- Сохранить текущее время
- В цикле вызвать несколько библиотечных функций N раз
  N = 100,1000,10000,100000,1000000
- Получить текущее время
- Вычислить время выполнения цикла
- Замеры производились на функциях strlen, memcpy, memcmp

Применимость статического метода

- Иногда при динамической линковке идет конфликт библиотек, важен порядок линковки. В таком случае можно часть библиотек отлаживать статически, часть динамически