Студент: Варичева Дарья Александровна

Группа: ИУ6-53Б

Вариант: 3 – Расчёт расстояний между точками

1. Цель работы

Выявить узкие места в сервисе B (ЛР2), измерить производительность, память, CPU, latency и оптимизировать реализацию.

2. Описание сервиса

– Сервис A: принимает значения координат точек и отправляет эти данные сервису B;

– Сервис B (неоптимально): вычисляет расстояние между точками по координатам и отправляет ответ сервису A.

В коде используются избыточное количество повторений одних и тех же операций, функции pow() и sqrt(), и операции с BigDecimal (эти функции и операции нагружают процессор), а также выполняются лишние преобразования типов (то есть boxing/unboxing). Листинг 1 с данным кодом сервиса B приведён ниже.

Листинг 1 – Код сервиса B до оптимизации

```
@Service
public class DistanceService {
    public Mono<Double> calculate(double x1, double y1, double x2,
double y2) {
        return Mono.fromCallable(() -> {
            BigDecimal result = BigDecimal.ZERO;
            MathContext ro = new MathContext(20,
RoundingMode.HALF_UP);
            for (int i = 0; i < 10000; i++) {
                BigDecimal bx1 = new BigDecimal(Double.toString(x1));
                BigDecimal by1 = new BigDecimal(Double.toString(y1));
                BigDecimal bx2 = new BigDecimal(Double.toString(x2));
                BigDecimal by2 = new BigDecimal(Double.toString(y2));
                BigDecimal dx = bx2.subtract(bx1, ro);
                BigDecimal dy = by2.subtract(by1, ro);
                Double boxedDx = Double.valueOf(dx.doubleValue());
```

Продолжение листинга 1

```
                Double boxedDy = Double.valueOf(dy.doubleValue());
                dx = new
BigDecimal(boxedDx.doubleValue());
                dy = new BigDecimal(boxedDy.doubleValue());
                BigDecimal dx2 = dx.pow(2, ro);
                BigDecimal dy2 = dy.pow(2, ro);
                BigDecimal sum = dx2.add(dy2, ro);
                result = sum.sqrt(MathContext.DECIMAL128);
            }
            return result.doubleValue();
        }).subscribeOn(Schedulers.boundedElastic());
    }
}
```

3. Профилирование до оптимизации

Во время проведения теста на сервис B было отправлено 200 запросов из Postman на вычисление расстояния между координатами. Результаты тестов показаны на рисунках 1-7.

– JDK Flight Recorder (JFR)

Рисунок 1 – Проверка CPU usage до оптимизации

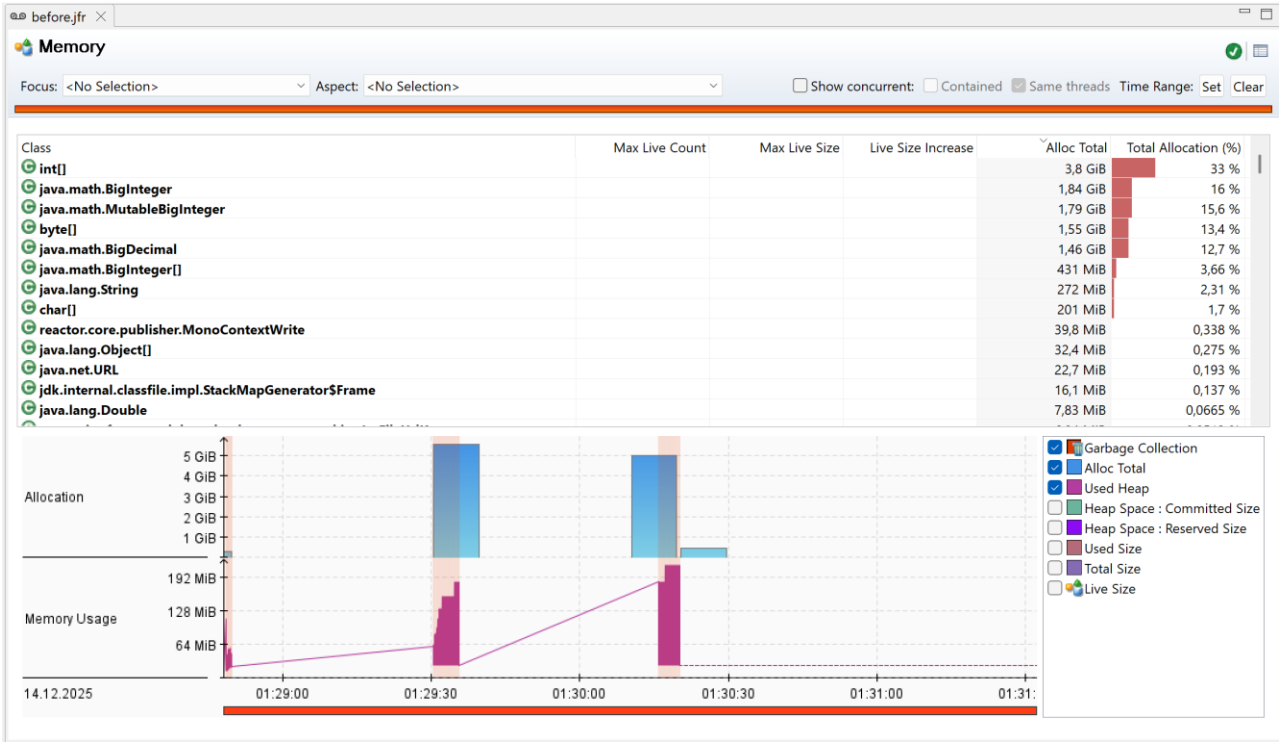

Рисунок 2 – Проверка allocation rate в Memory до оптимизации

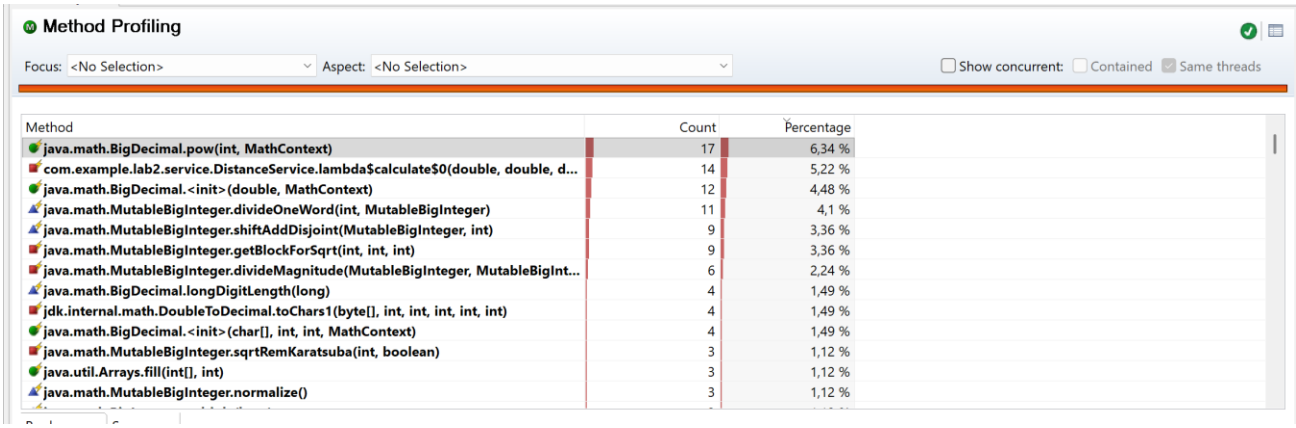

Рисунок 3 – Проверка hot spots в Method Profiling до оптимизации

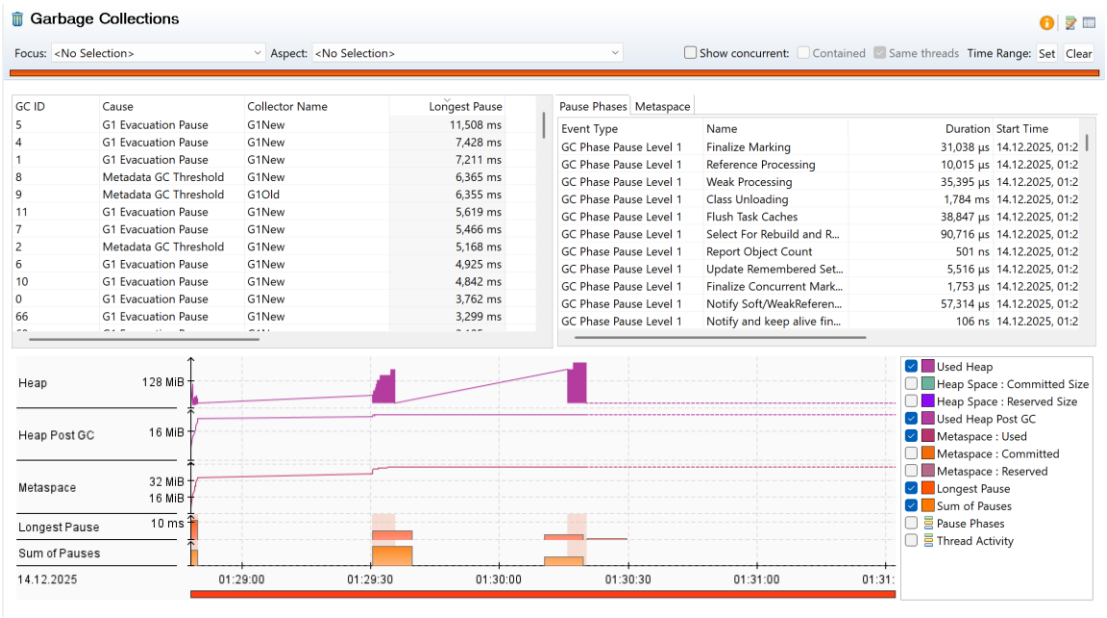

Рисунок 4 – Проверка pause times в GC до оптимизации

– Async Profiler (CPU)

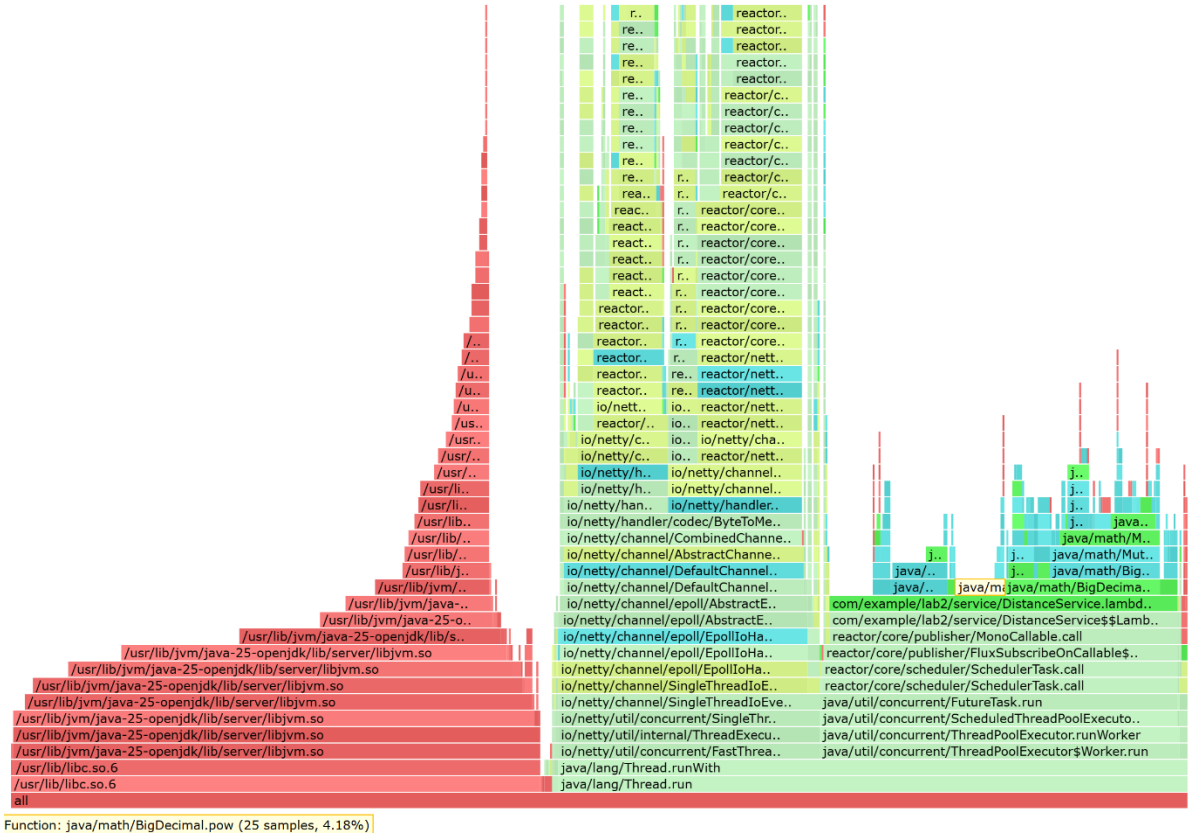

Рисунок 1 – Flame graph (CPU) с выделенной функцией pow() до оптимизации

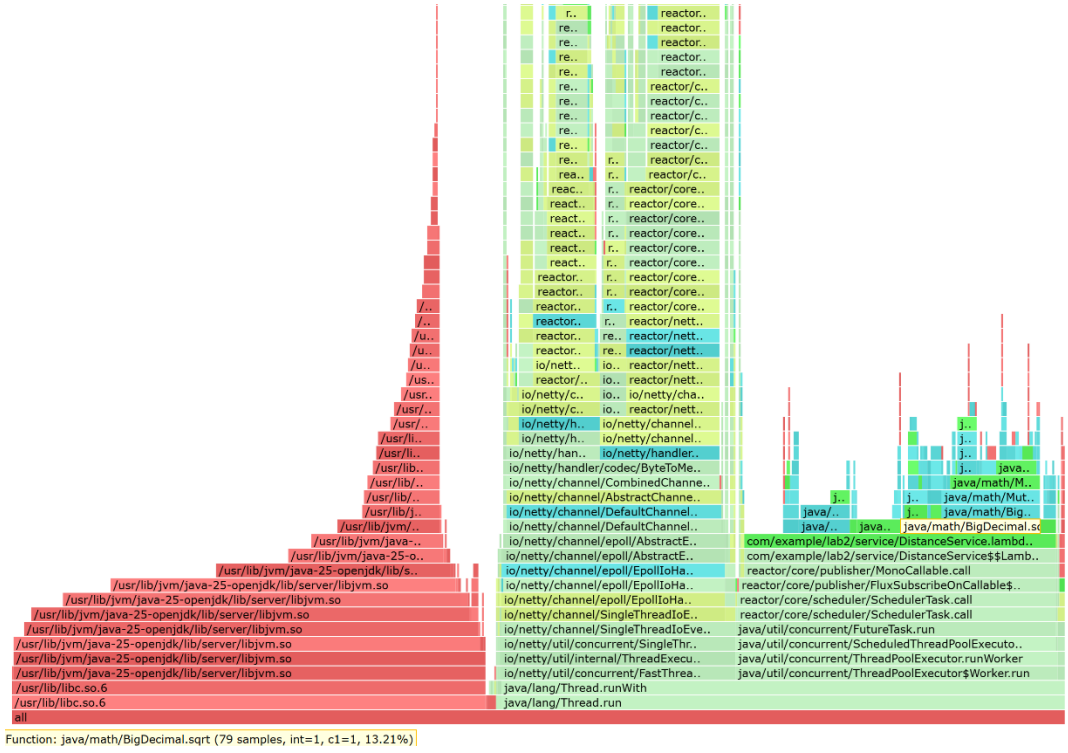

Рисунок 2 – Flame graph (CPU) с выделенной функцией sqrt() до оптимизации

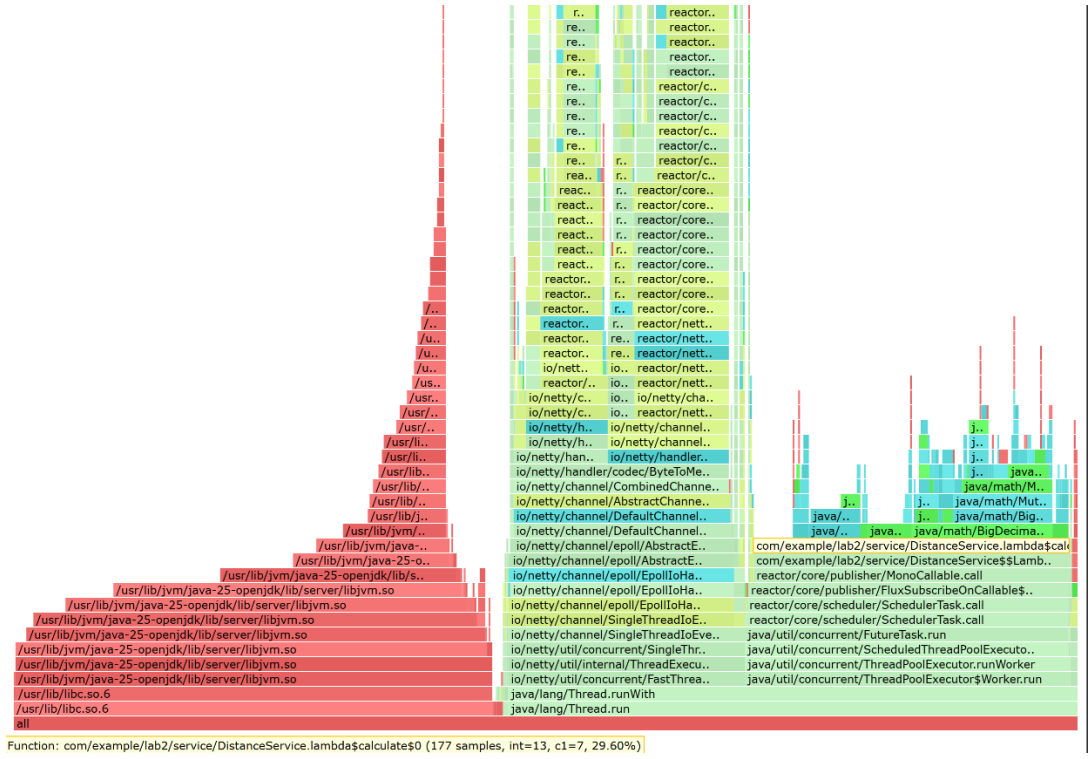

Рисунок 3 – Flame graph (CPU) с выделенной основной для сервиса B функцией calcucate() до оптимизации

4. Оптимизация

Чтобы оптимизировать код сервиса B, заменим тип BigDecimal на double, уберём цикл for и избыточные преобразования (boxing/unboxing), а также вместо функции pow(2, x) запишем операцию x*x. Все эти изменения отражены в листинге 2.

Листинг 2 – Код сервиса B после оптимизации

```
@Service
public class DistanceService {
    public Mono<Double> calculate(double x1, double y1, double x2,
double y2) {
        return Mono.fromCallable(() -> {
            double dx = x2 - x1;
            double dy = y2 - y1;
            double distance = Math.sqrt((dx * dx) + (dy * dy));
            return distance;
        }).subscribeOn(Schedulers.boundedElastic());
    }
}
```

5. Профилирование после оптимизации

Во время проведения теста на сервис B было также отправлено 200 запросов из Postman на вычисление расстояния между координатами. Результаты тестов показаны на рисунках 8-12.

– JDK Flight Recorder (JFR)

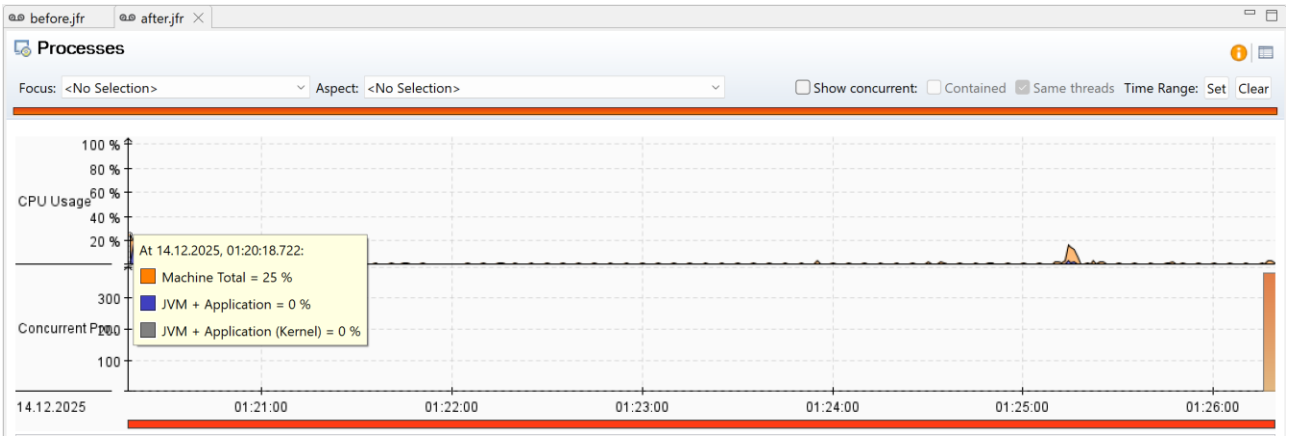

Рисунок 8 – Проверка CPU usage после оптимизации

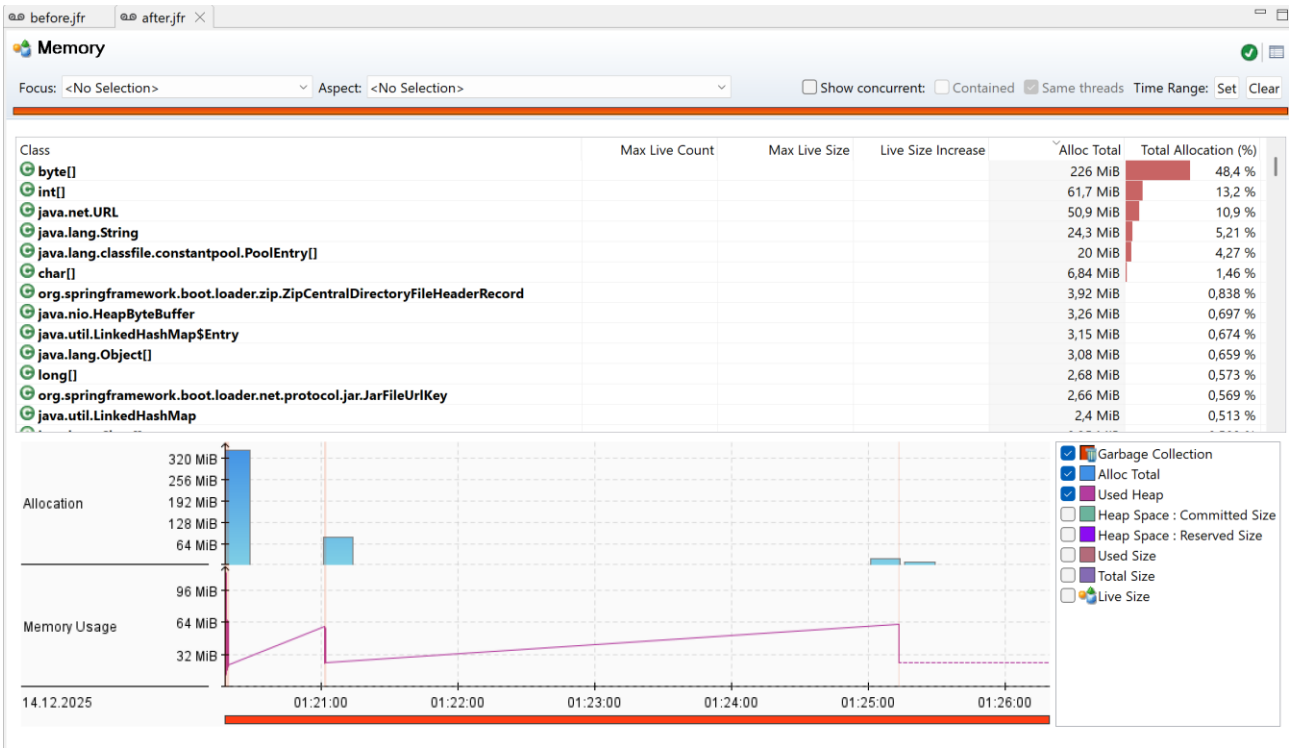

Рисунок 9 – Проверка allocation rate в Memory после оптимизации

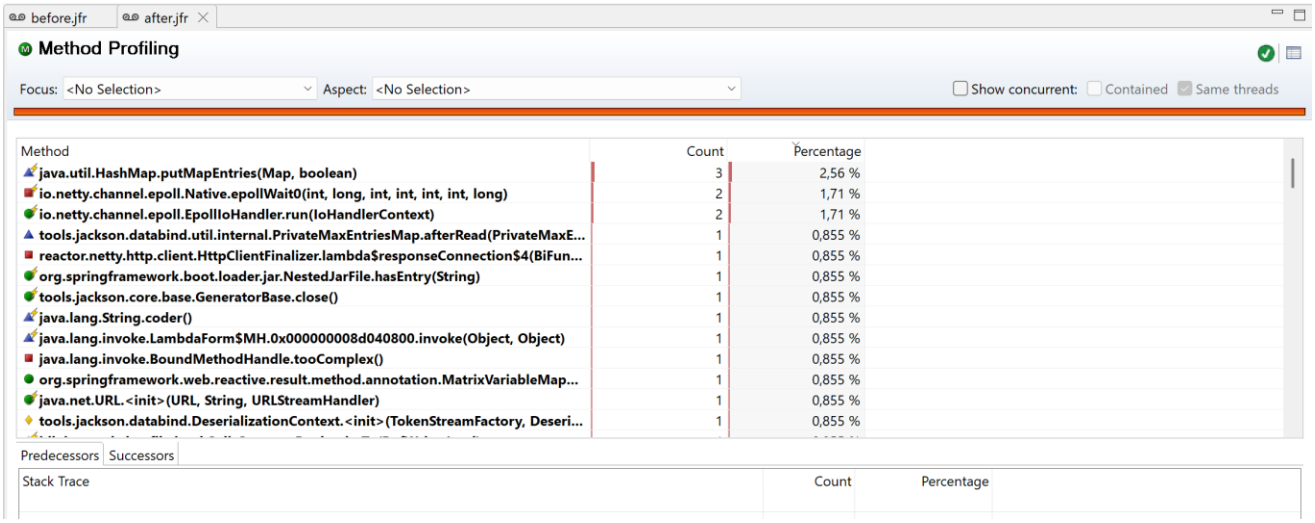

Рисунок 10 – Проверка hot spots в Method Profiling после оптимизации

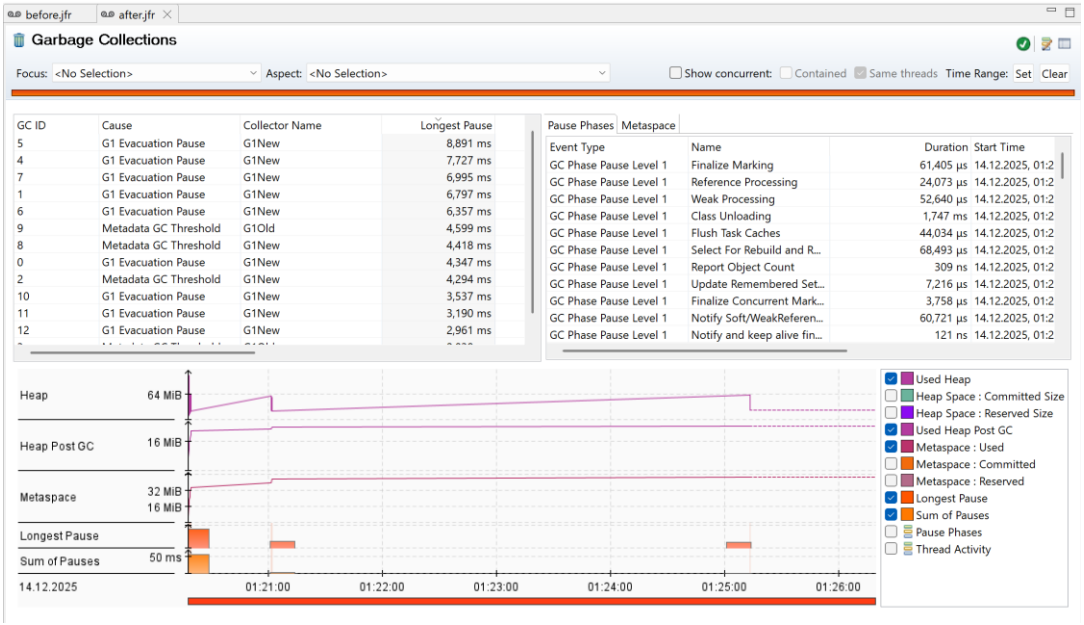

Рисунок 4 – Проверка pause times в GC после оптимизации

– Async Profiler (CPU)

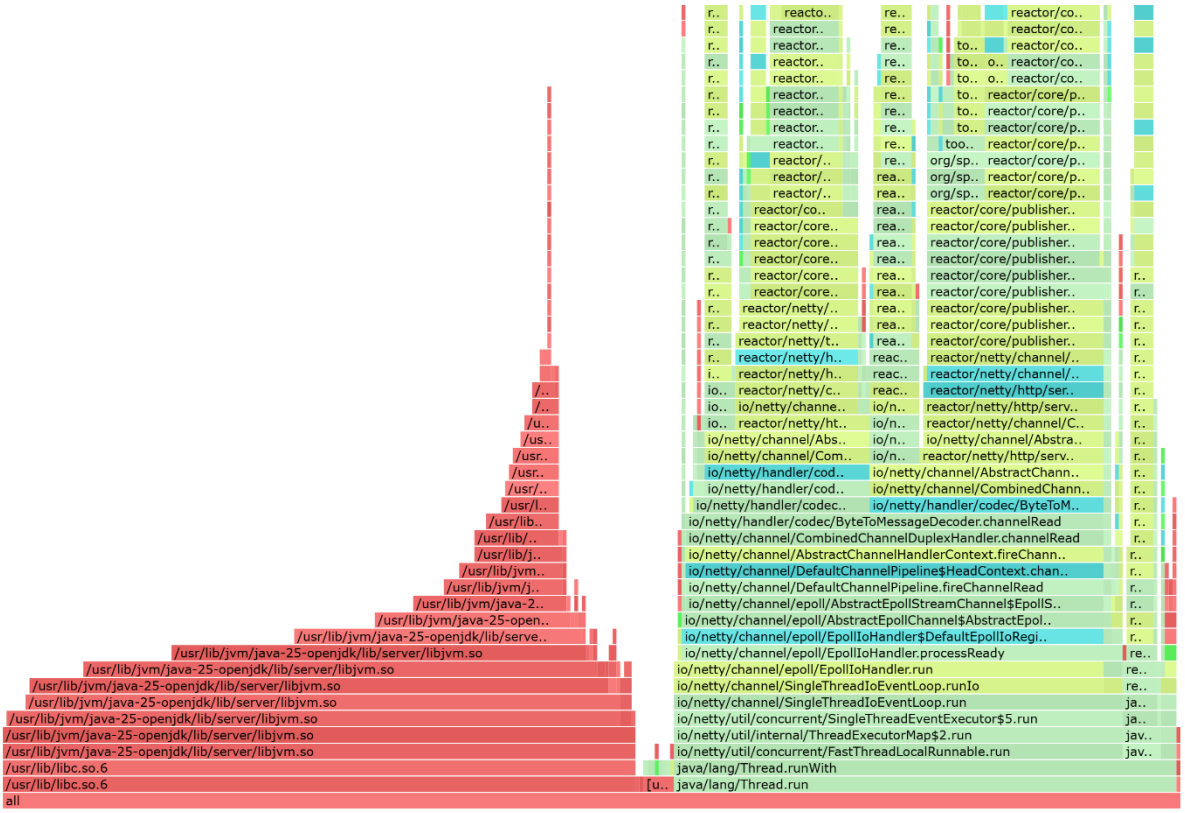

Рисунок 5 – Flame graph (CPU) после оптимизации

Согласно Flame graph (CPU) на рисунке 12, нагрузка на процессор от основной функции calculate() сервиса B настолько мала, что информация на ней не отражается в данной диаграммме. В таблице 1 продемонстрированы результаты тестов до и после оптимизации.

Таблица 1 – Результаты тестов

Метрика	Значение до оптимизации	Значение после оптимизации
CPU usage	33%	25%
Allocation rate	3,8 Гб/с	226 Мб/с
Latency	13,2 мс (из Postman)	9,4 мс (из Postman)
GC pause time	11,5 мс	8,8 мс

Вывод

Оптимизация кода для сервиса B позволила уменьшить нагрузку на процессор, объём памяти, выделяемый под временные переменные, и время работы сборщика мусора, а также ускорить выполнение запросов.