Отчет по практике № 1

Измерение производительности приложения посредством анализа использования ЦП

Рассмотрены следующие задачи:

* Сбор данных об использовании ЦП
* Анализ данных о загрузке ЦП

### Шаг 1: Сбор данных профилирования

1. Открыв проект в Visual Studio мы установили 2 точки останова, одну на 51 строке кода, место откуда нужно сделать проверку, а вторую на 199 строке в конце функции.

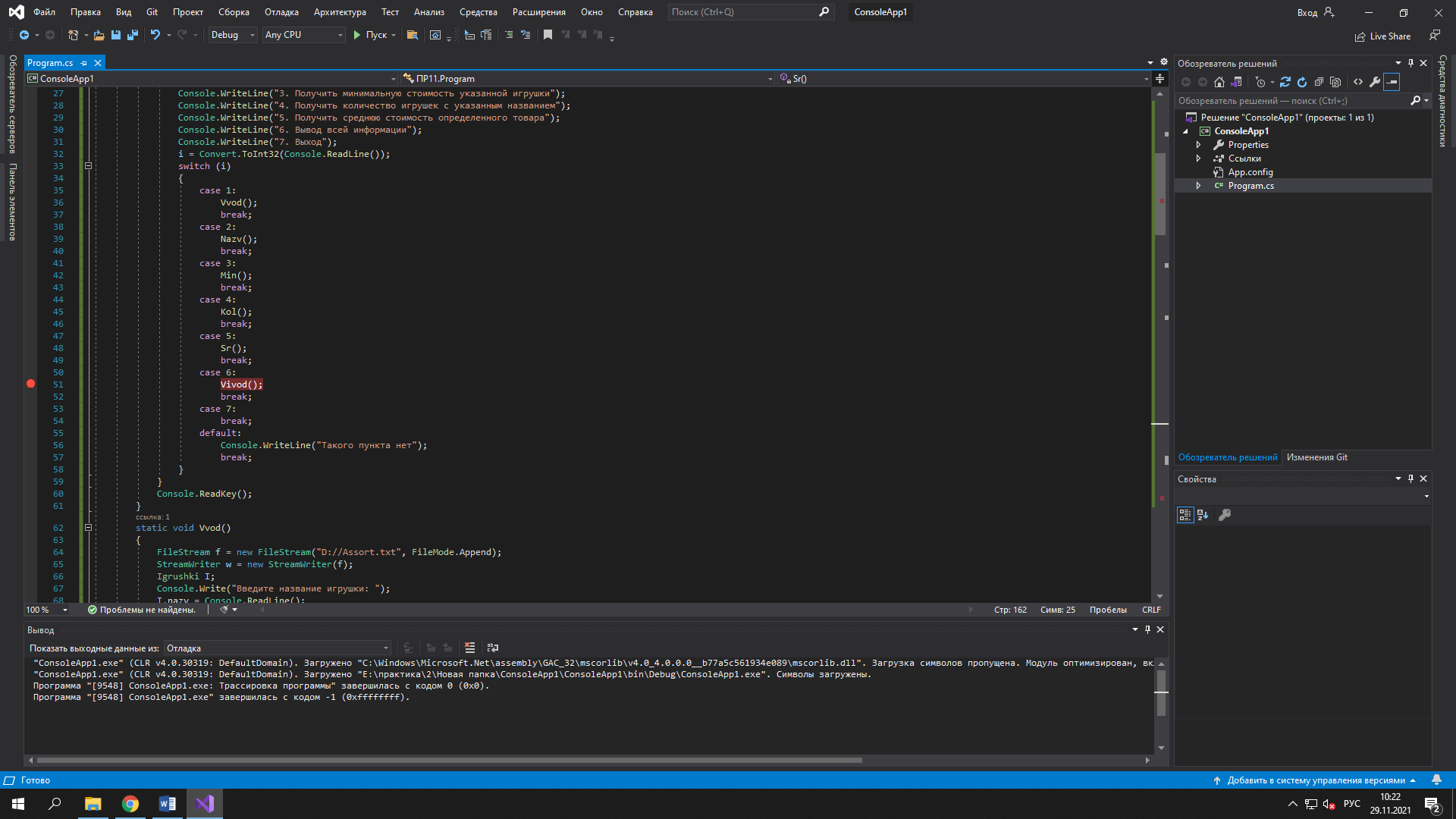


Рис.

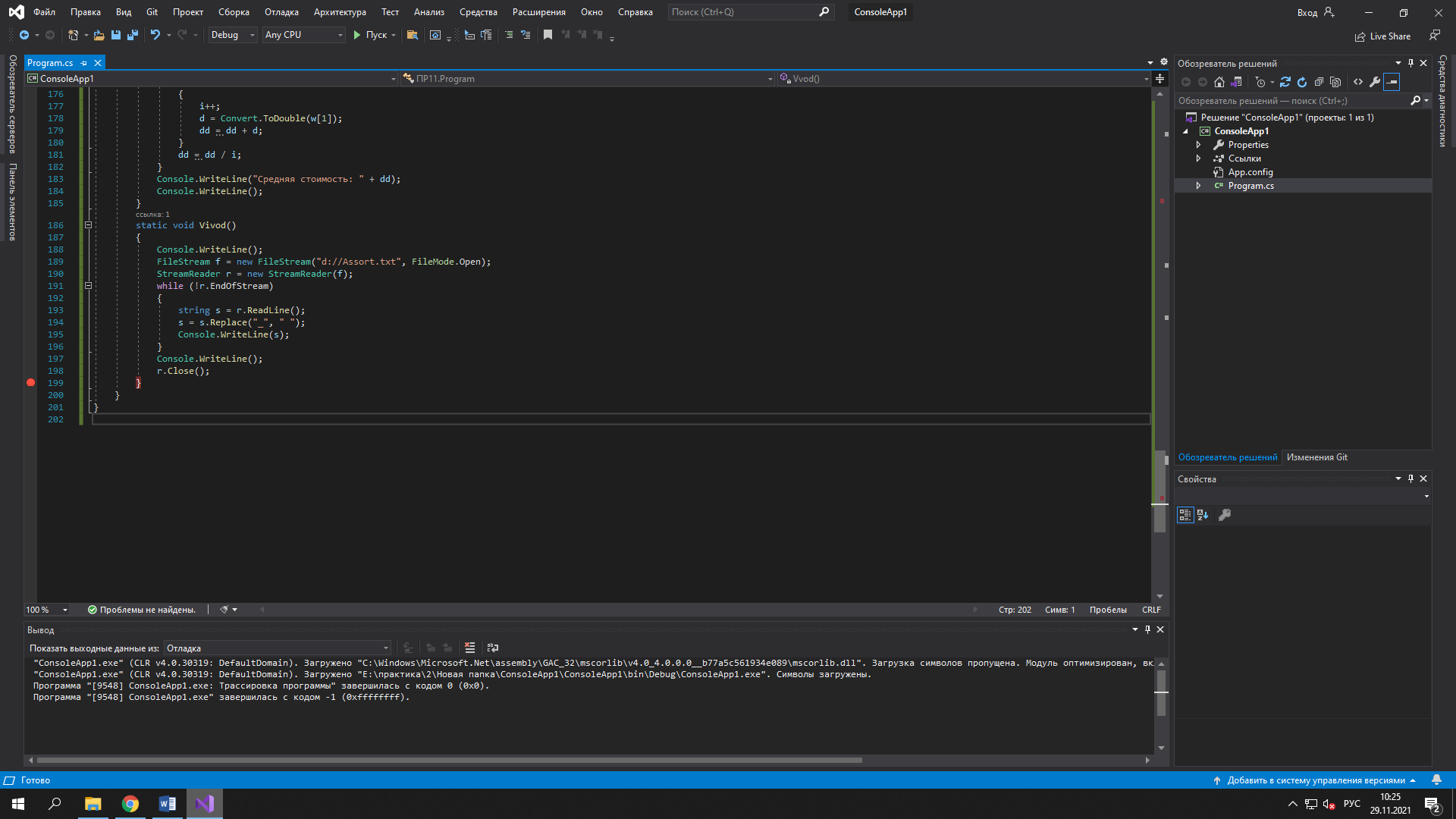


Рис.

1. После этого мы открываем окно средства диагностики

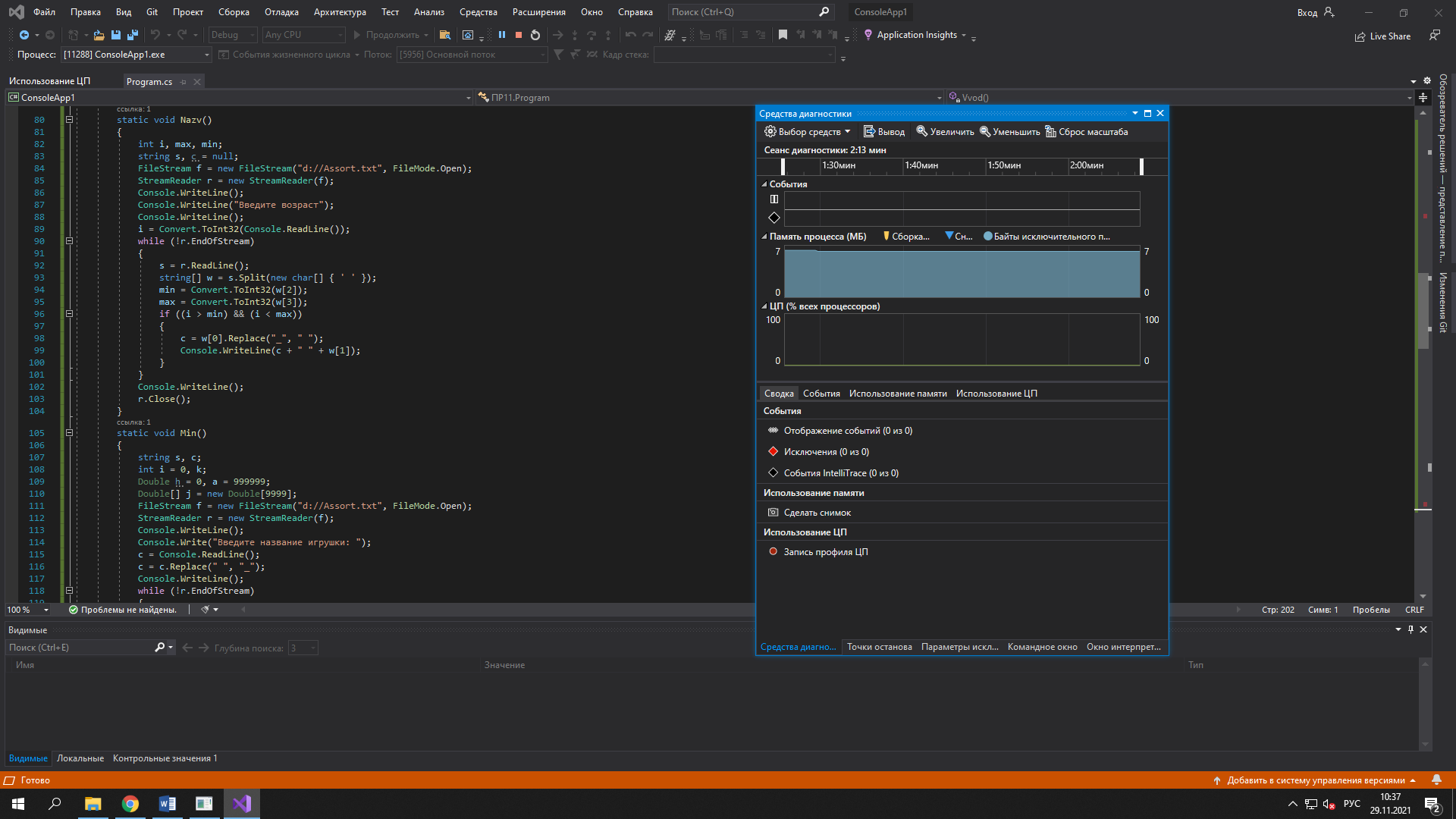


Рис.

1. После этого выбираем средства анализа

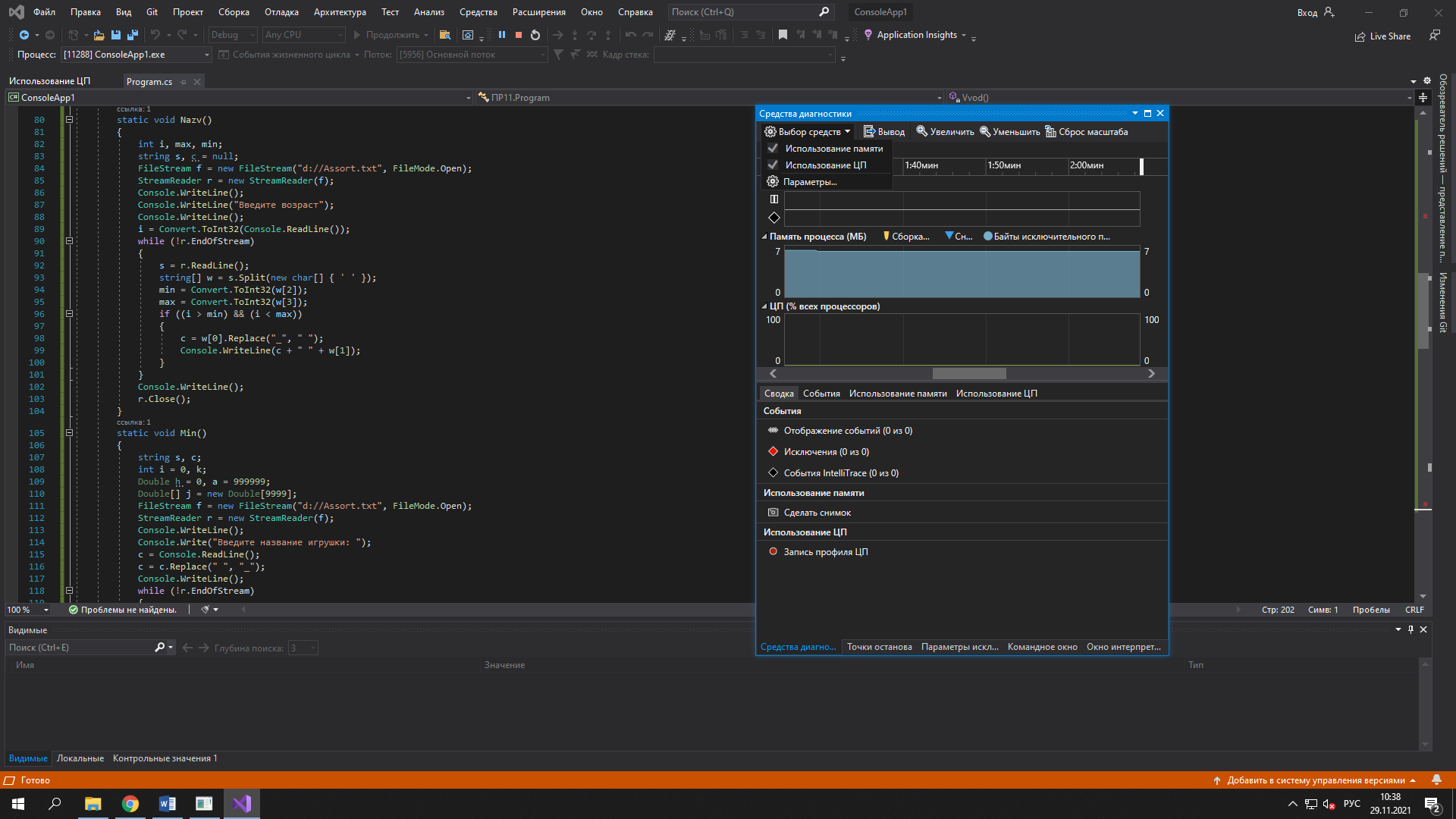


Рис.

1. Запускаем отладку и начинаем запись профиля ЦП и вызываем событие первой точки останова

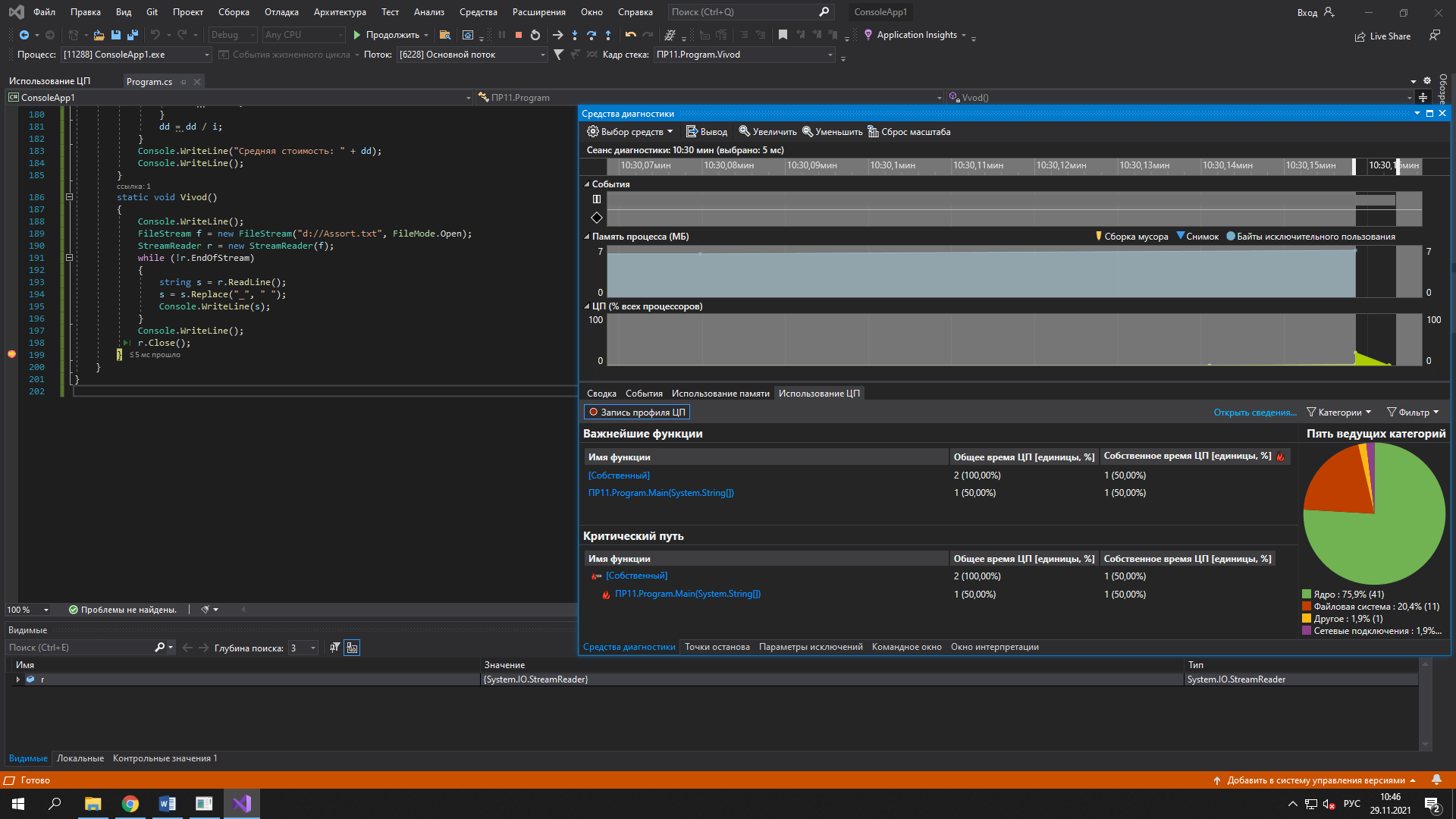


Рис.

**Шаг 2:** Анализ данных о загрузке ЦП

1. Открываем функции

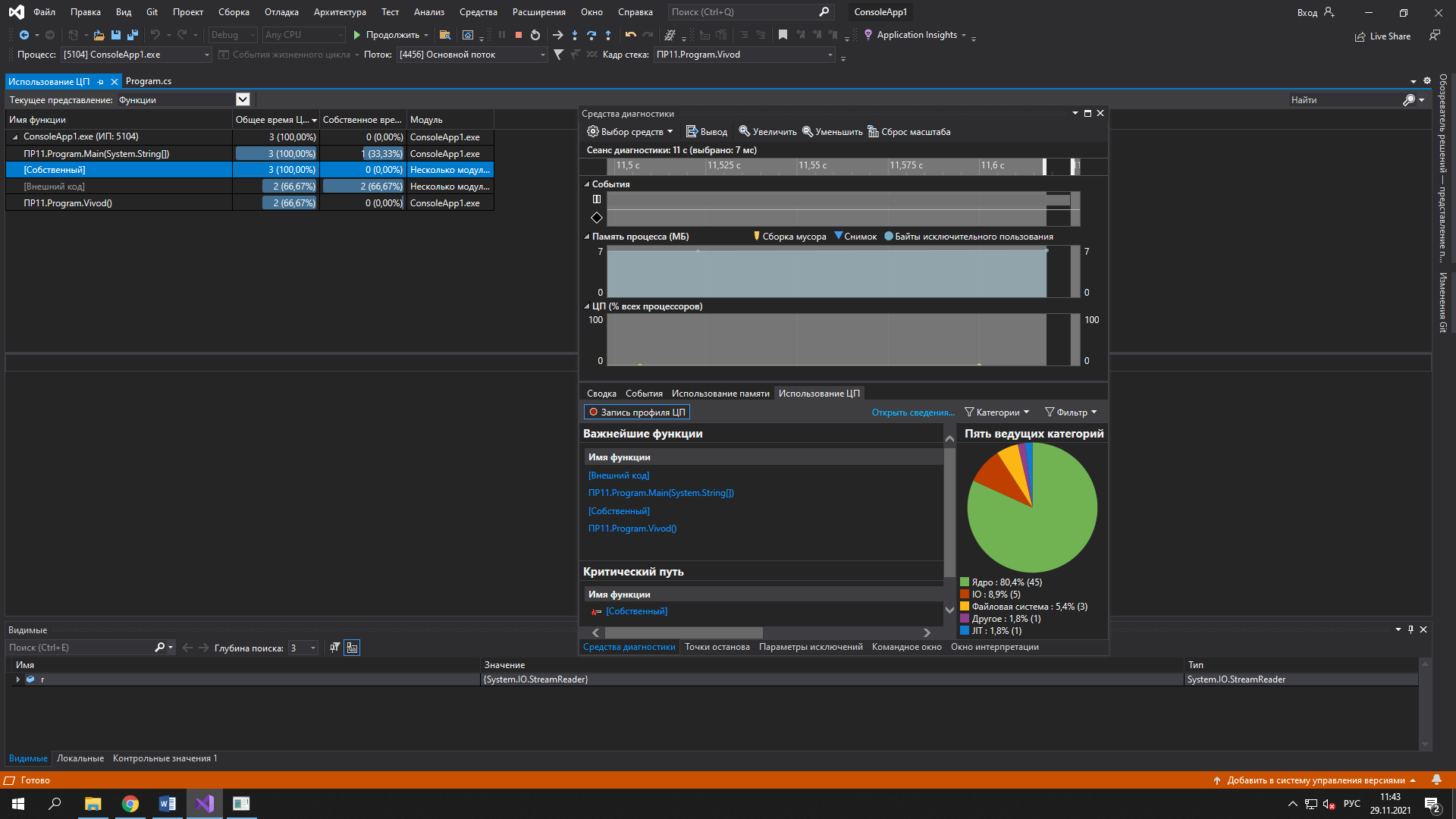


Рис.

1. Смотрим на самую загруженную функцию

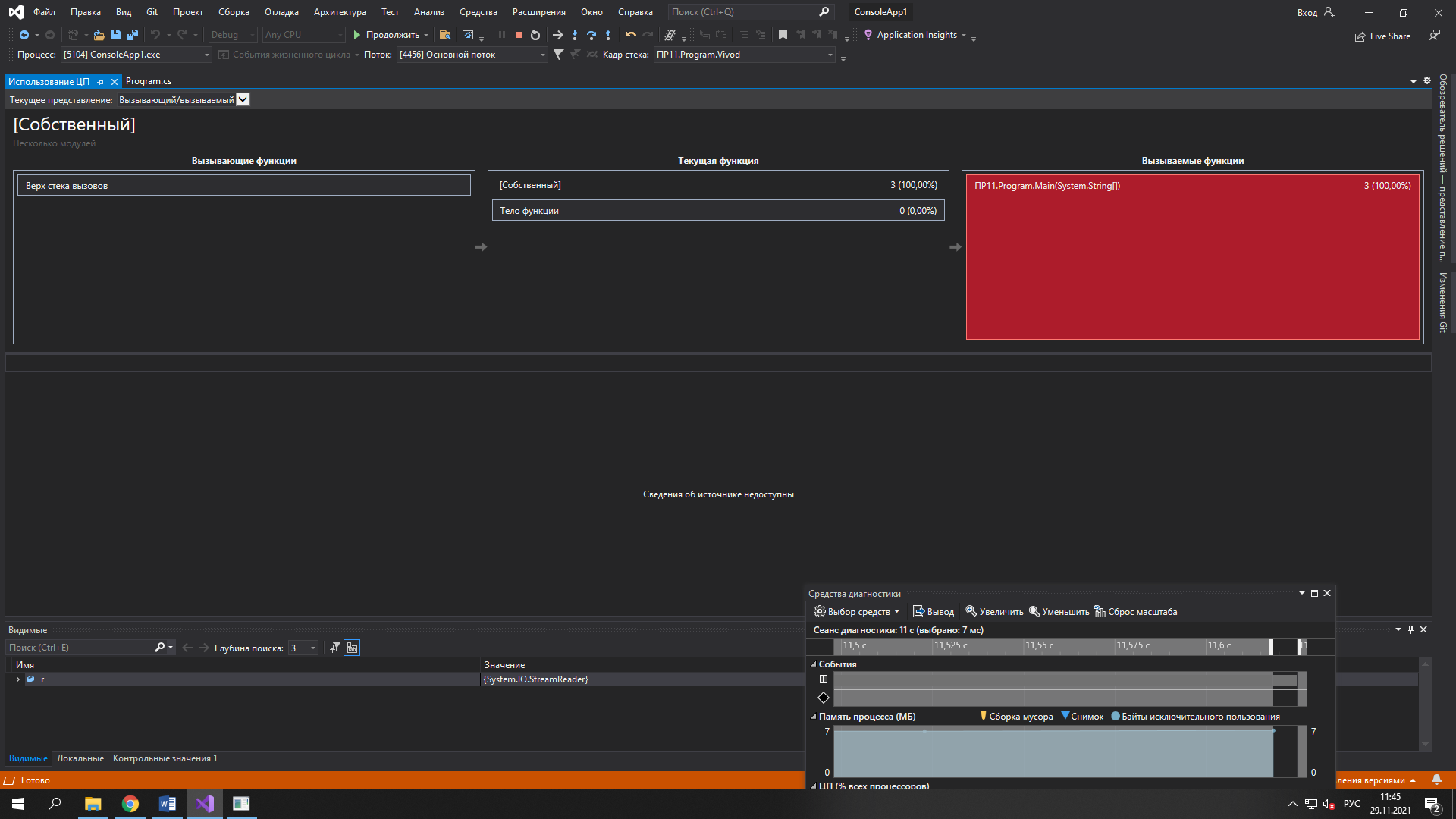


Рис.

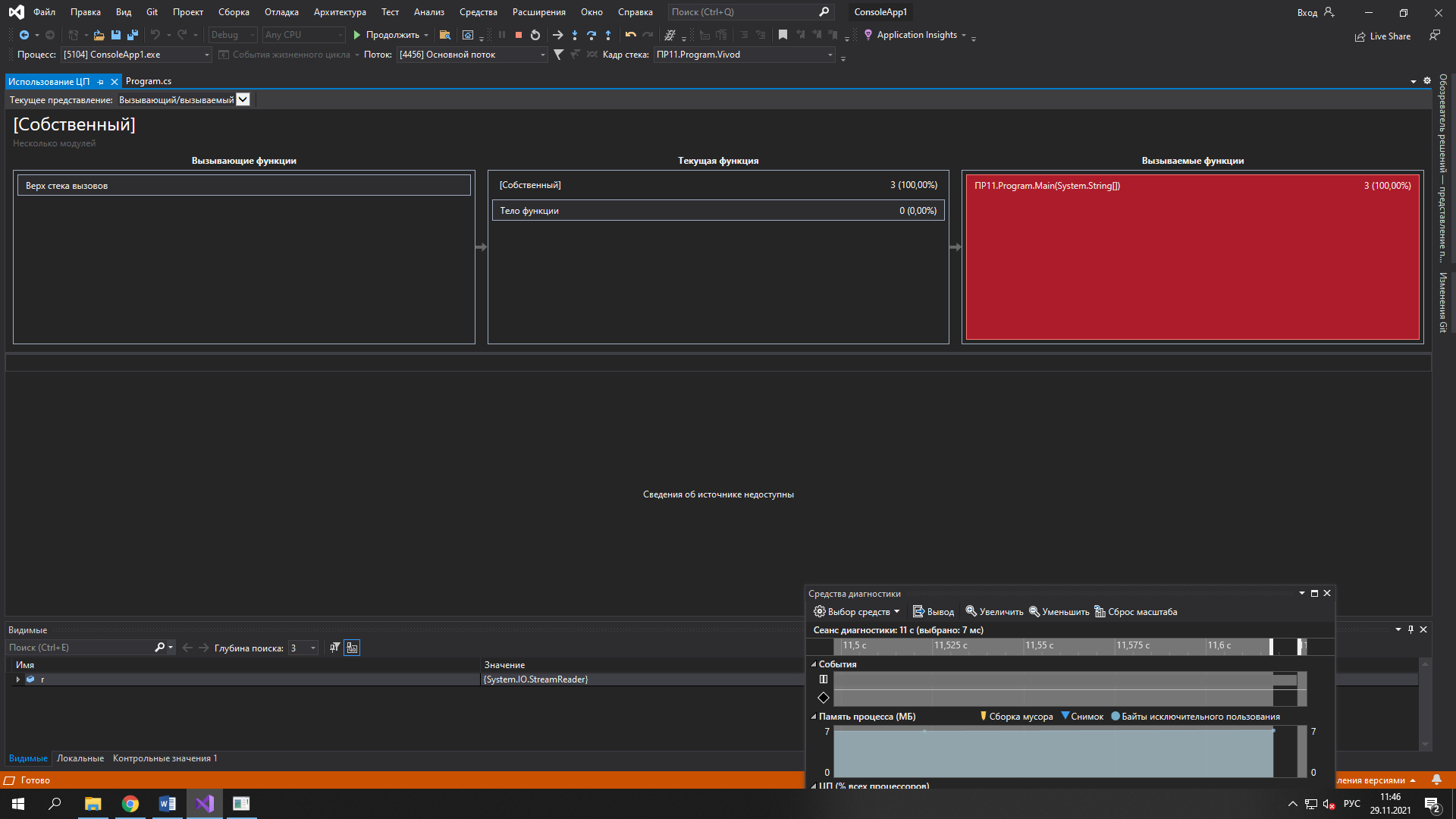


Рис.

1. Рассмотрим дерево вызовов

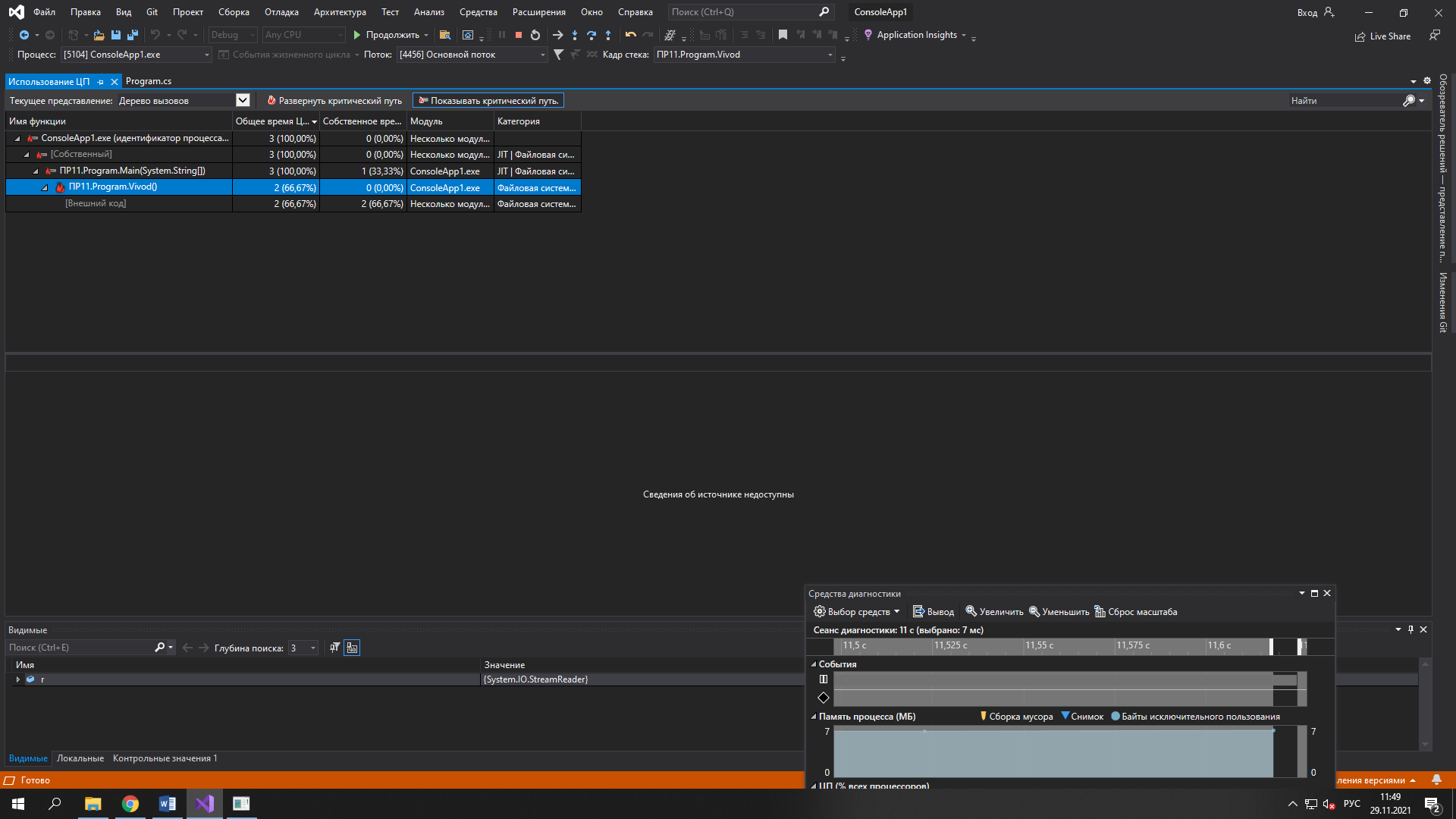


Рис.

Подводя итог анализа, можно сказать что проблемы с производительностью находятся в модуле ConsoleApp1.exe. Развернув критический путь, мы видим что функция Vivod() использует самый высокий процент ЦП.

**Упрощение кода**

На строках 83 и 109 убираем неиспользуемые переменные

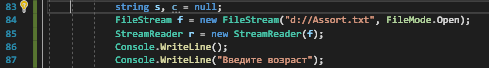


Рис.10 Строка с неиспользуемой переменной

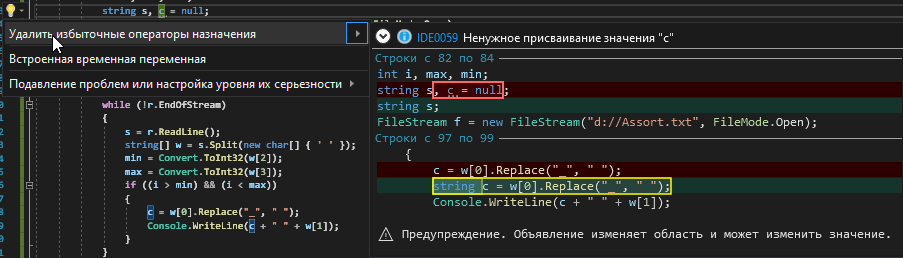


Рис.11 Удаление избыточного оператора



Рис.12 Строка после упрощения

На строках 179 и 181 используем составной оператор значения



Рис.13 Строка без упрощения



Рис.14 Строка с составным оператором



Рис.15 Строка без упрощения



Рис.16 Строка с составным оператором

Выполняем диагностику упрощенного кода:

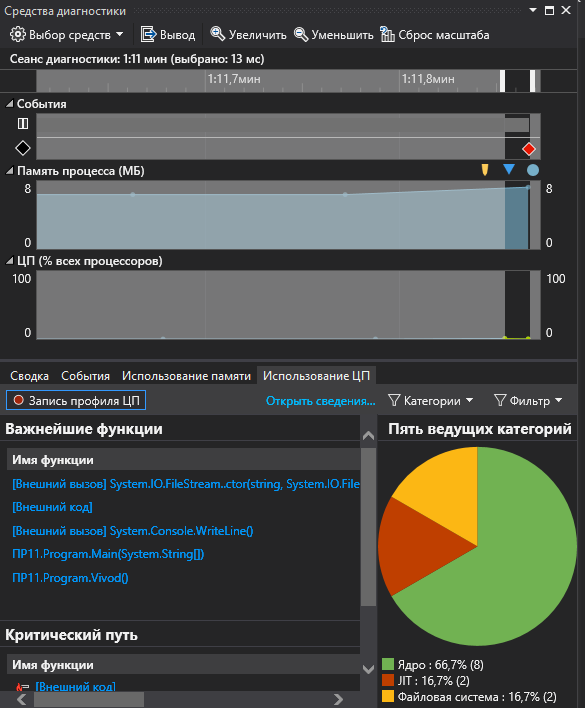


Рис.17 Средства диагностики

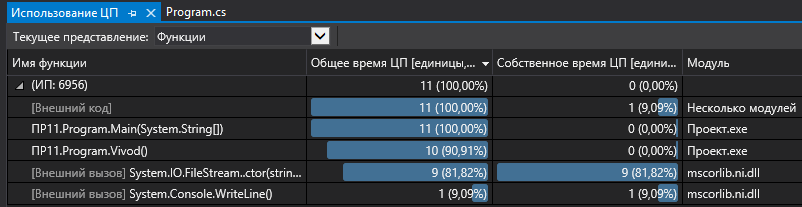


Рис.18 Функции

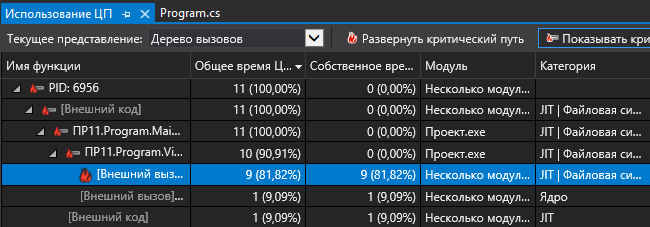


Рис.19 Дерево вызовов

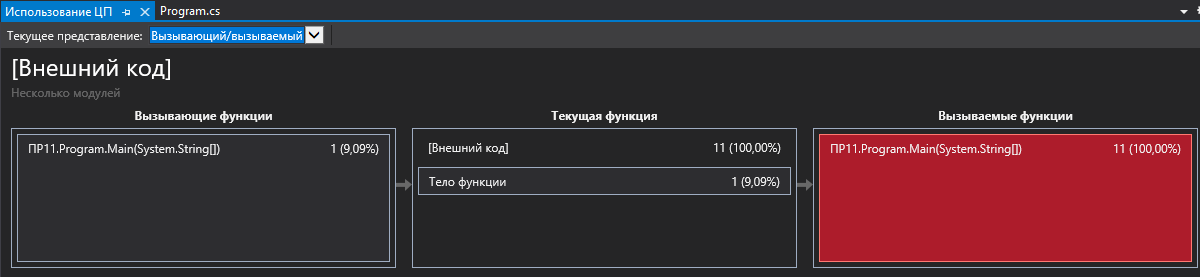


Рис.20 Вызывающий/вызываемый

**Вывод:** Как мы видим после упрощения кода спала нагрузка с ядра и файловой системы. Уменьшилось собственное время ЦП, что свидетельствует о том, что были решены некоторые проблемы внутри самой функции.