**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕЬЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА(ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**По практической работе**

**По дисциплине «Параллельные алгоритмы и системы»**

Выполнил

Студент гр.9891 Палунин А.И.

Преподаватель Пазников А. А.

Санкт-Петербург

2022

**Цель работы:**

1. Собрать тесты из пакета SPEC CPU 2017

2. Запустить тесты

3. Провести анализ эффективности с помощью утилиты perf (cashe-misses);

4. Найти места в коде, где больше всего промахов.

**Содержание:**

* Тест nab\_r 3
* Тест deepsjeng 4
* Тест exchange2 7
* Тест cactuBSSN 9

Из пакета были скомпилированы следующие тесты:

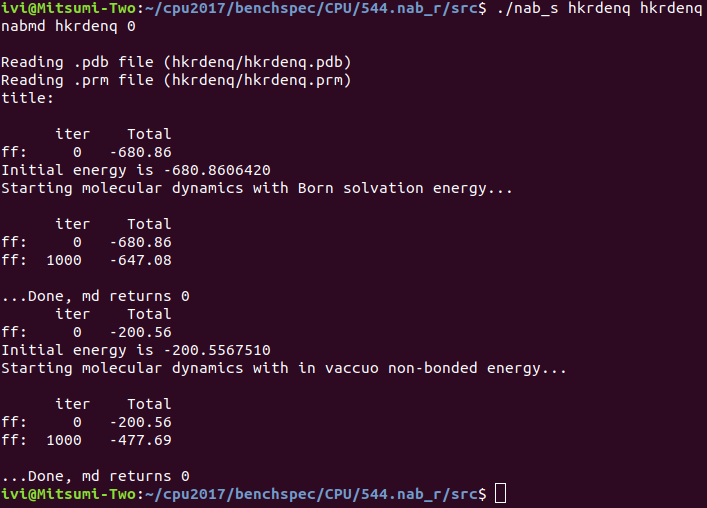
- **nab\_r** (файл, запускающий тест находится в директории /src/nab\_s). Сборка была осуществлена с помощью скрипта находящегося в этой же папке: simple-build-nab-s-644.sh. Для запуска теста в аргументах командной строки указываем наименование тестовых файлов (они расположены в каталоге src/hkrdenq). Результат работы теста представлен на рисунке 1.

Рисунок 1. Запуск теста nab\_r

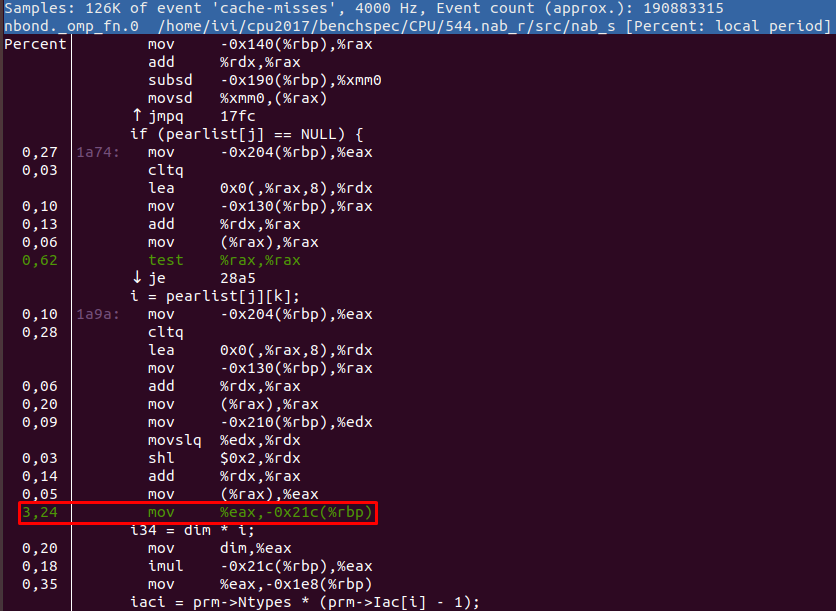
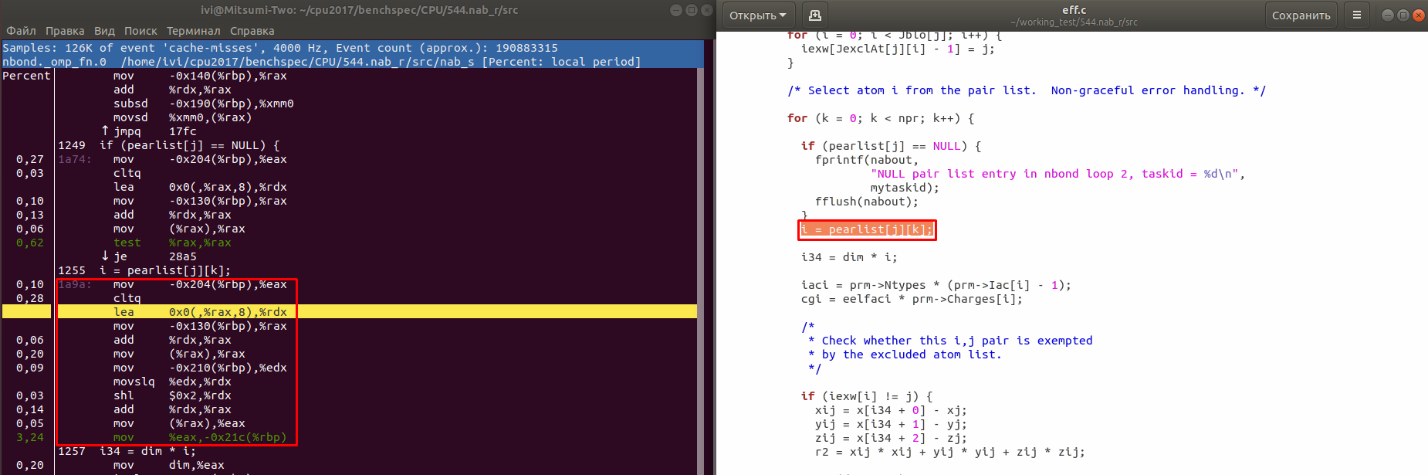
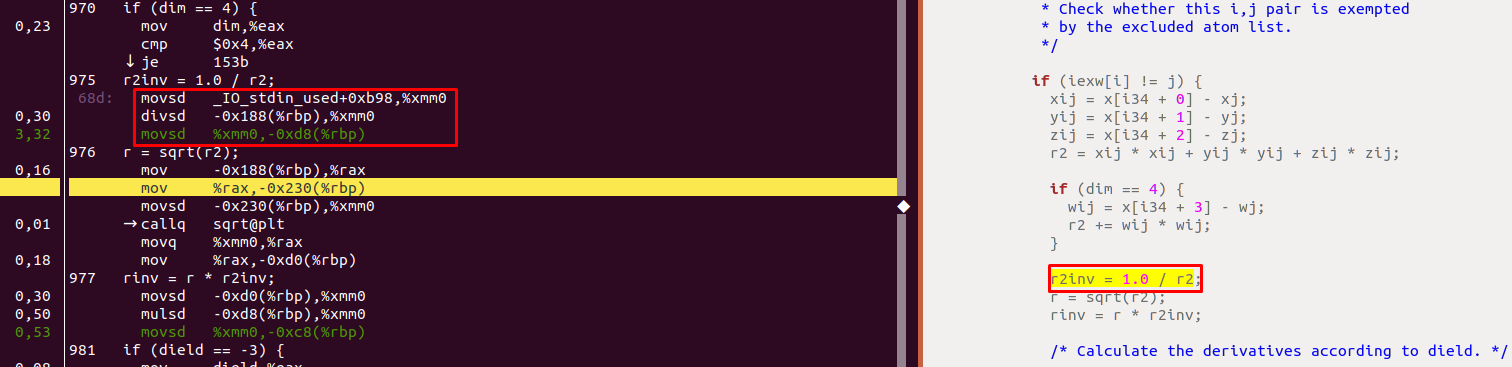
Результат работы утилиты perf представлен на рисунке 2. Наибольшее число промахов в данном тесте присутствует на команде mov %eax, -0x21c(%rbp). Найдем в исходных файлах теста код, соответствующий этой команде – рисунок 3.

Рисунок 2.

Рисунок 3.

Как видно из кода, промахи присутствуют в момент присвоения переменной i значения из массива pearlist.

Ещё промахи присутствуют при выполнении копирования одного участка памяти в виде двойных слов (DWORD) в другой, об этом нам говорит ассемблерная команда movsd. Она участвует в вычислении переменной r2inv = 1.0/r2. Рисунок 4.

Рисунок 4.

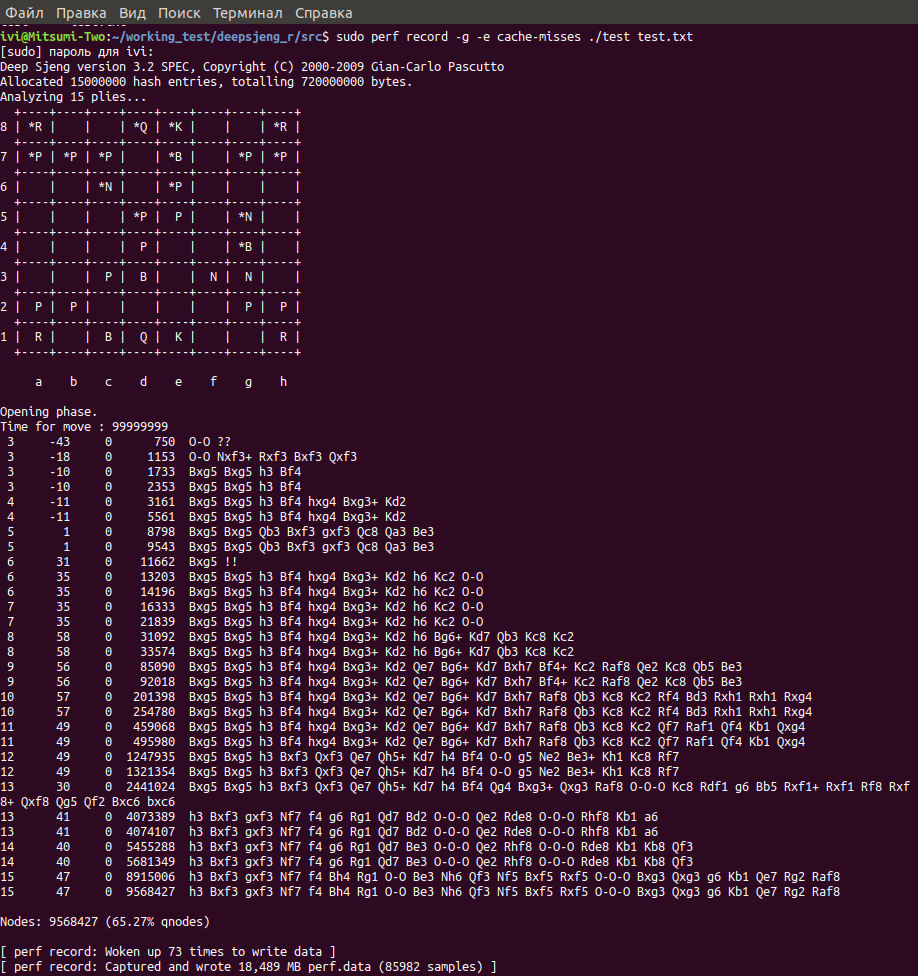
Больше в данном тесте не было найдено большого количества промахов.

Следующим был скомпилирован тест:

**- deepsjeng**. Сборка теста была осуществлена с использованием команды gcc -g \*.c -o test. Запускающим файлом является файл test. В качестве аргумента при запуске тест принимает файл test.txt в котором содержится шахматная позиция и глубина анализа позиции. Результат работы теста представлен на рисунке 5.

Результат работы утилиты perf представлен на рисунке 6. Как видно из рисунка cache-misses чаще всего встречается в search и qsearh. Посмотрим на их ассемблерный код и найдем сответствующие участки кода в исходных файлах.

Сначала проверим search\_root (рисунок 7). Как видим из ассемблерного кода проблемы возникают при изменении знака числа. Этому соответствует в исходном файле search.cpp функция rootmovesearch().

Рисунок 5. Результат работы теста deepsjeng

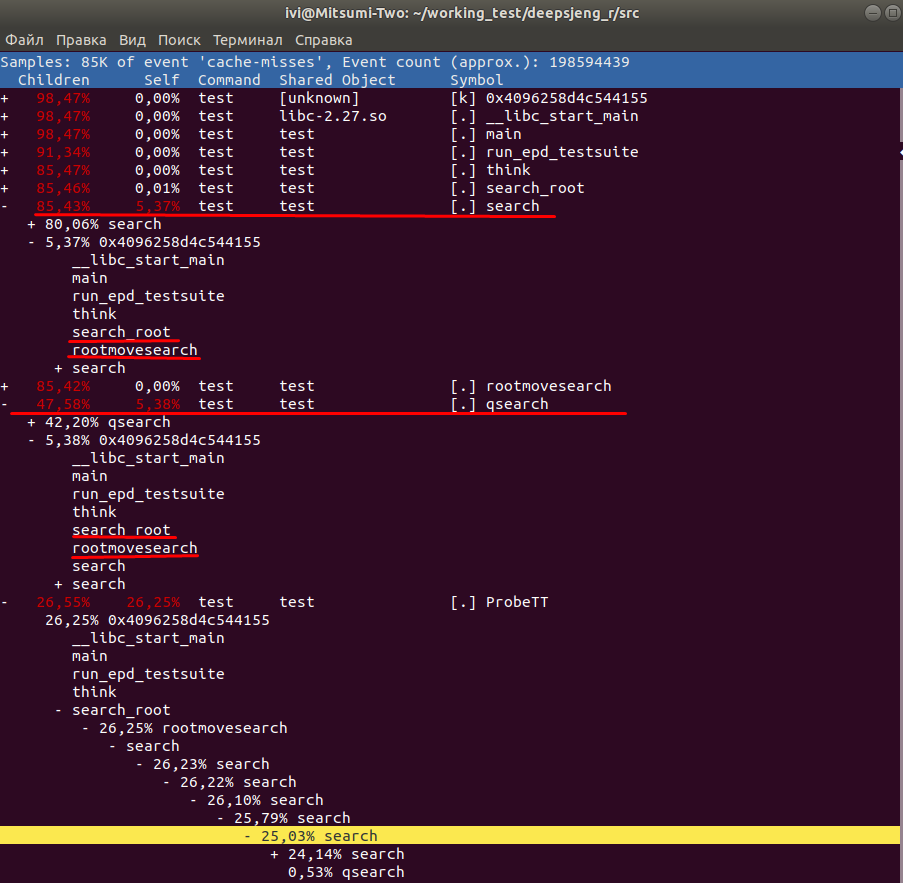
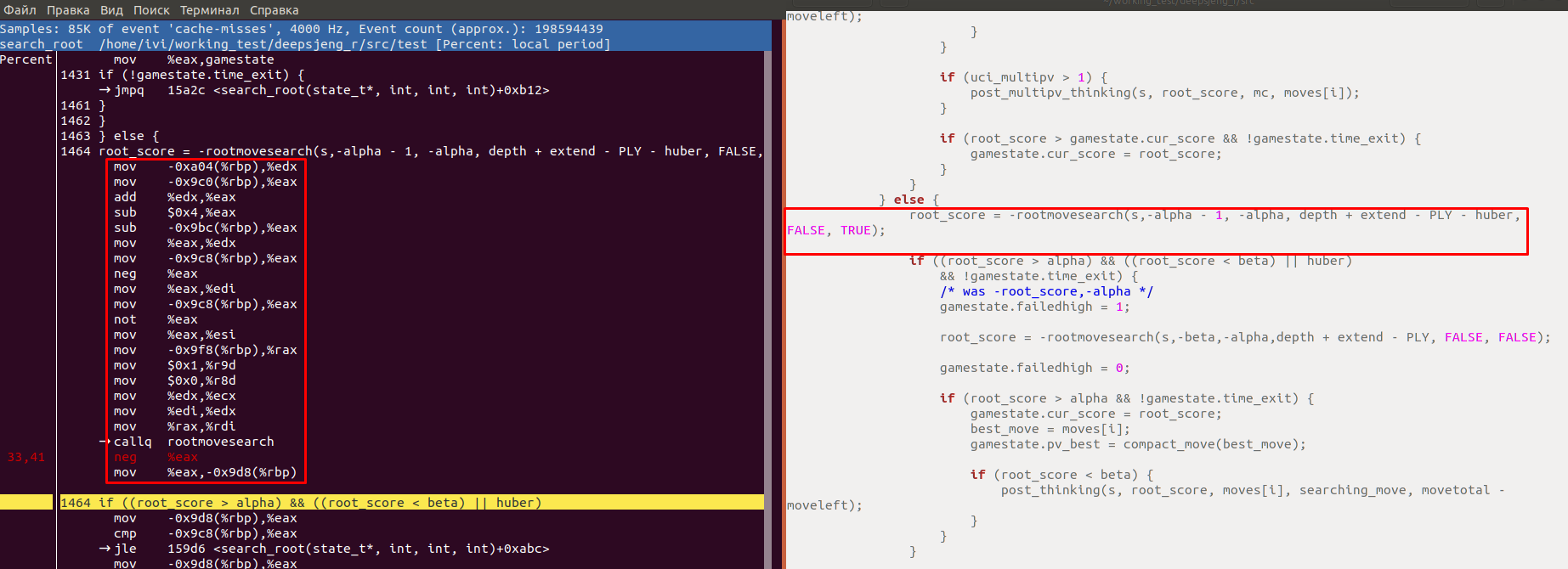
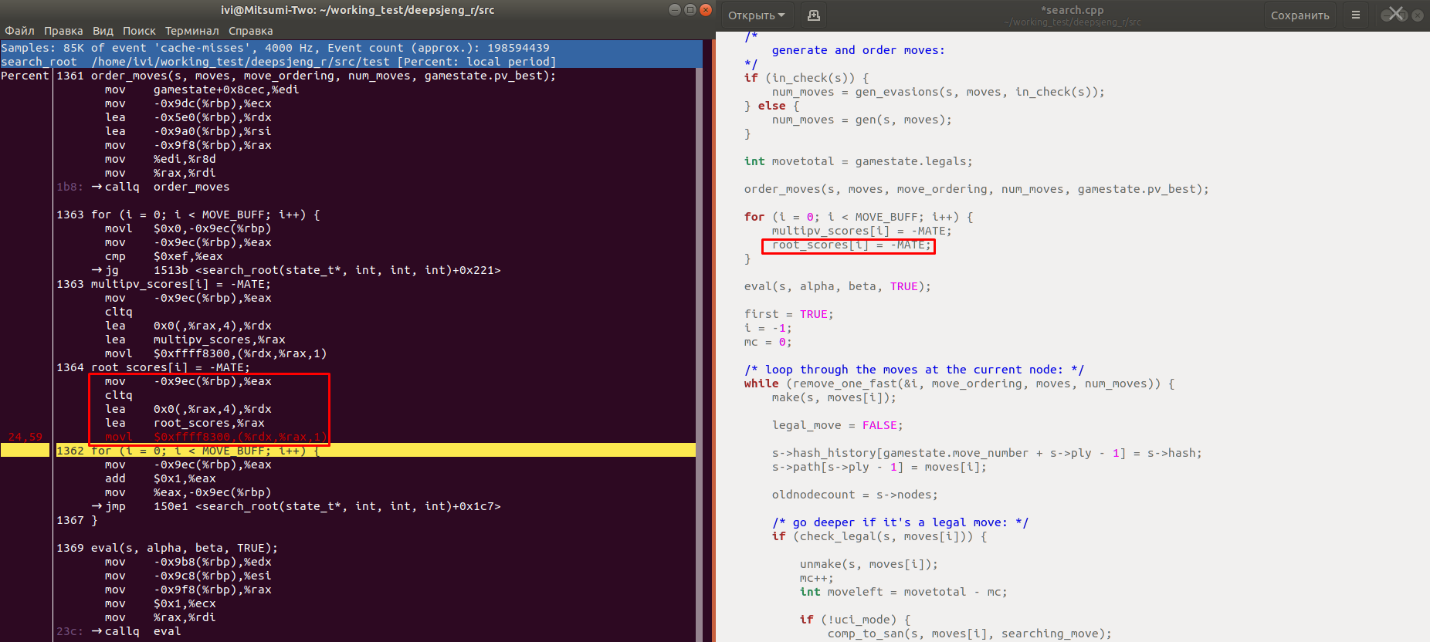


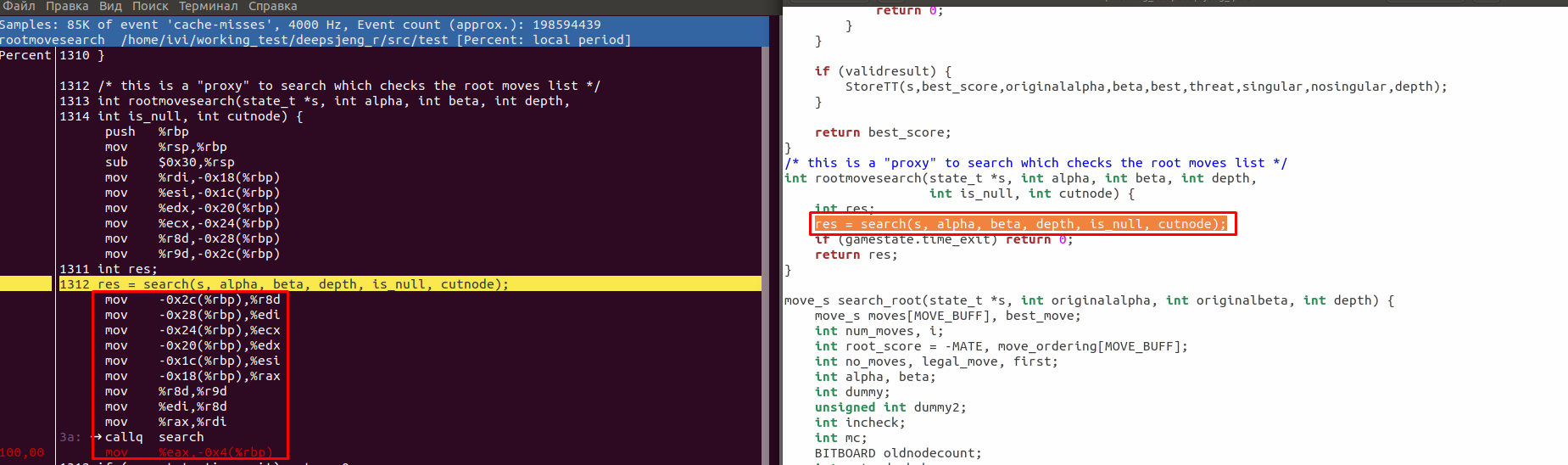
Рисунок 6. Результат работы утилиты perf report.

Рисунок 7.

На рисунке 8 видим проблему в том же файле search.cpp при работе цикла for(i=0;i<MOVE\_BUF;i++).

Рисунок 8.

Также в файле search.cpp присутствует проблема в функции rootmovesearch() в моменте присвоения переменной res значения из функции search(). Рисунок 9.

Рисунок 9.

Остальные cache-misses, которые видно на рисунке 6 находятся в файле search.cpp в функциях rootmovesearch() и search().

Следующим был скомпилирован тест:

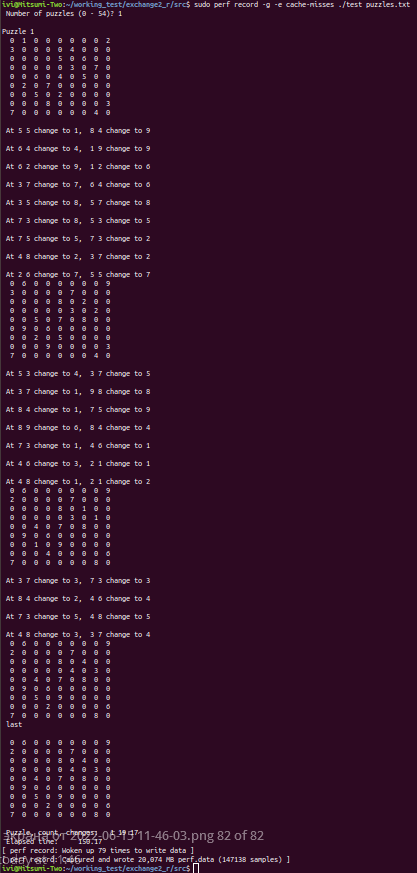
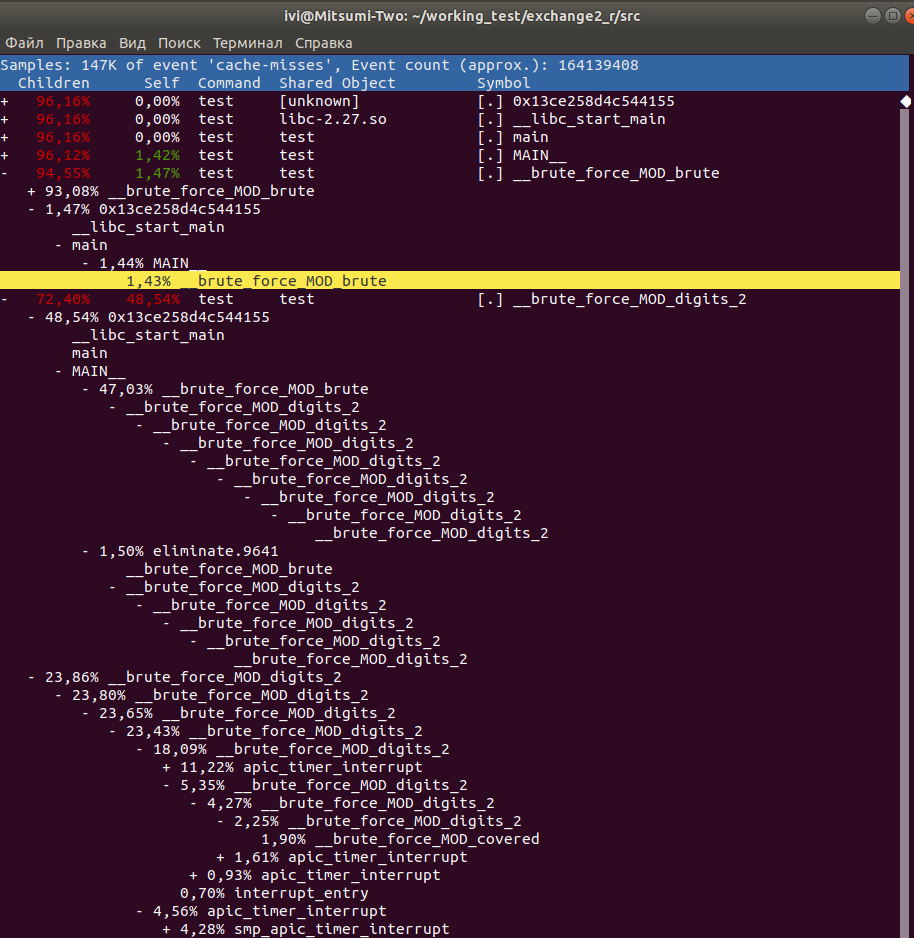
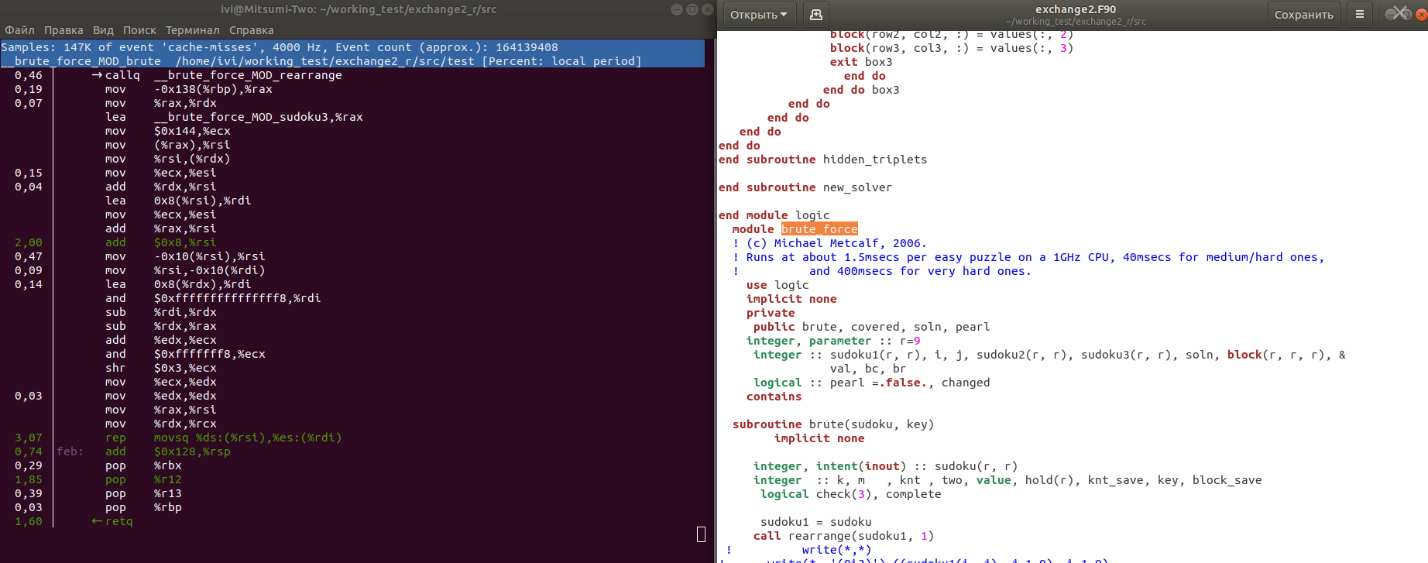
**- exchange2\_r.** Тест был скомпилирован командой gfortran -g \*.F90 -o test. Запускающим файлом является test. При запуске теста в качестве аргумента принимается файл (puzzles.txt), содержащий набор судоку. Результат работы теста представлен на рисунке 10.

Рисунок 10. Результат работы теста exchange2\_r

На рисунке 11 изображен отчёт, полученный в результате работы утилиты perf. Исходя из отчёта можно сделать вывод, что чаще всего cache-misses встречается в \_\_brute\_force\_MOD\_digits\_2. Посмотрим аннотированный код и найдем соответствующие участки кода в исходных файлах.

Рисунок 11. Отчёт perf.

В данном тесте нет серьезных cache-misses, исходя из аннотированного кода (рисунок 12). Небольшие cache-misses встречаются в модуле brute\_force в файле exchange2.F90

Рисунок 12. Аннотированный код и модуль brute\_force.

Следующим был скомпилирован тест:

**- cactuBSSN**. Компиляция выполнялась утилитой simple-build-cactuBSSN\_s-607.sh. Запускающий файл cactuBSSN\_s. При запуске, в качестве аргумента указываем файл spec\_test.par. На рисунке 13 изображен результат работы данного теста.

Далее запустим утилиту perf report, чтобы увидеть где есть cache-misses. На рисунке 14 изображен report сформированный утилитой perf.

Исходя из отчёта в тесте чаще всего встречаются cache-misses в op\_real\_update\_2.\_omp\_fn.6 и в GOMP\_parralel, но GOMP\_parralel является сторонней библиотекой (GNU libgomp) и поэтому её рассматривать не будет. Посмотрим на аннотированный код op\_real\_update\_2.\_omp\_fn.6 рисунок 15.

Найдем файл, в котором присутствуют cache-misses. Для этого будем использовать утилиту grep с аргументом рекурсивного поиска по каталогу исходных файлов теста. Файл в котором присутствует проблема Operators.c. Промахи кеша случаются при вычислении значения по формуле scale\*varptr[i]+fact0\*srcptr[i]+fact1\*srcptr[i] и записи этого значения в массив varptr[] (Рисунок 15).

Рисунок 13. Результат работы теста cactuBSSN

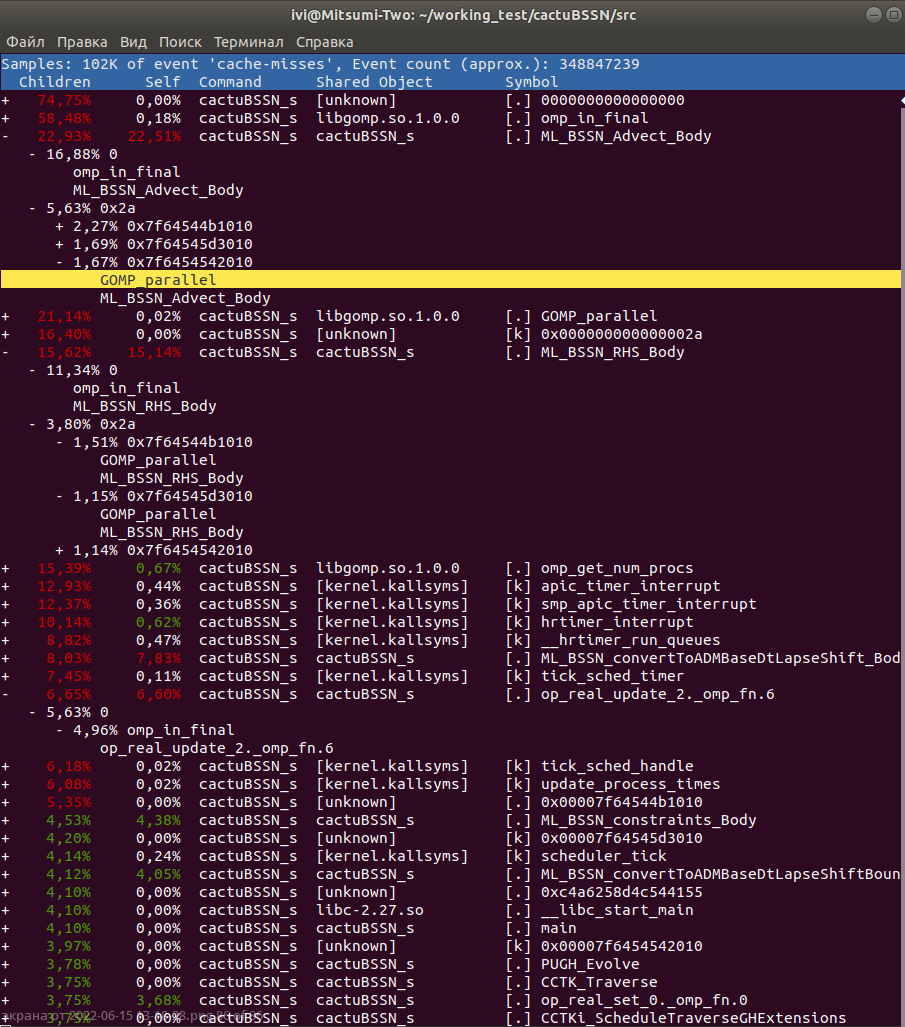
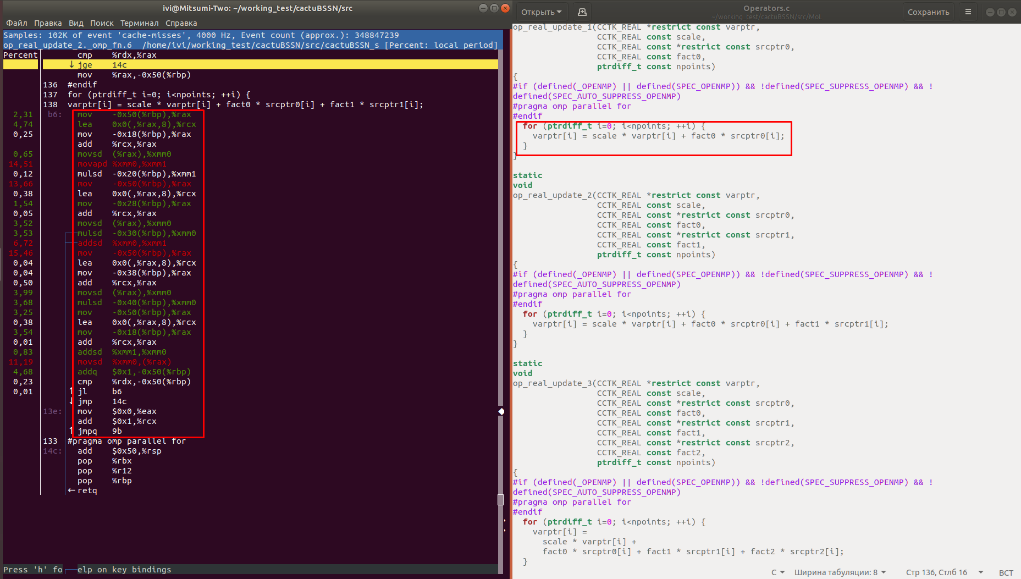
Рисунок 14. report сформированной утилитой perf

Рисунок 15. Аннотированный код и проблемные cache-misses