МЦСТ

Устройство современных компиляторов

Лекция 4.
Инструменты анализа кода.

Маркин А. Л.
alexanius@gmail.com

Часть 1.
Исследование исполняемого файла.

Исследование исполняемого файла

Иногда возникает необходимость исследовать содержимое компилируемого приложения или изучить уже полученный бинарный файл.

Такая необходимость может возникнуть при появлении неполадок или при анализе действий компилятора.

Системы GNU/Linux предоставляет богатый набор инструментов, позволяющих проанализировать получаемый файлы.

Исследование исполняемого файла

В самой простой ситуации у нас есть исходник, и мы можем посмотреть его содержимое на различных стадиях компиляции.

- `gcc -v` — обеспечивает подробный вывод запускаемых программ с переменными окружения.
- `gcc -E` — выведет файл после препроцессирования.
- `gcc -S` — выведет получаемый ассемблер.
- `gcc -fdump-tree-all` — сбросит представление программы

Исследование исполняемого файла

Если исходного кода нет, или нужно понять что по факту получилось после компиляции. то можно использовать следующие команды:

- `file` — определяет тип файла
- `readelf` — печатает информацию об исполняемом файле
- `ldd` — печатает список зависимостей от общих пакетов
- `nm` — печатает символы, определённые в файле
- `strings` — выводит все последовательности печатных символов файла
- `objdump` — дизассемблер. Переводит исполняемый файл в ассемблерные инструкции

Часть 2.
Оценка производительности системы.

Оценка производительности системы

При выборе машины для дальнейших вычислений следует понимать какой производительностью она обладает. Для измерения производительности существует много различных тестов, из которых реальную ситуацию отображают лишь несколько.

При работе с суперкомпьютерами принято измерять производительность на вычислительных приложениях. Распространёнными тестами являются:

- **hpl** — решение системы линейных уравнений вида $Ax = b$.
- **hpcg** — производительность работы с данными на примере задачи обработки разреженных матриц.

Оценка производительности системы

За что мы боремся?

При грамотном подходе к оптимизациям производительность может увеличиваться на порядок. Пример производительности сокращённого теста **hpl** при разных линейках оптимизации:

- `-O0`: 689
- `-O1`: 2723
- `-O2`: 2810
- `-O3`: 4396

Часть 2.
Профилирование приложения.

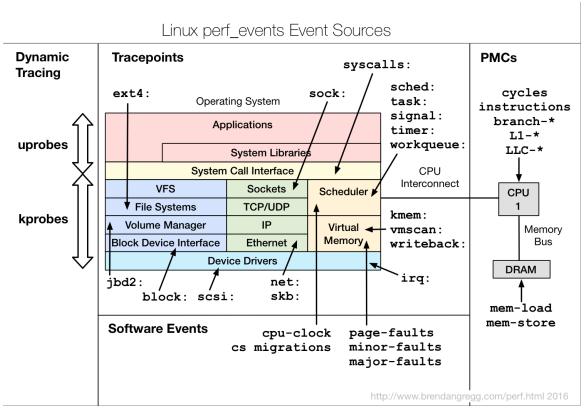
Профилирование приложения

Профилирование — это процесс сбора информации о приложении. Собирать можно различную информацию: вызовы функций, время исполнения различных участков, обращения к файлам, работу с сетью и т.д.

Профилирование приложения позволяет понять характер работы программы, увидеть её горячие и холодные участки, оценить наличие «бутылочных горлышек» и мест, которые следует исправить.

Профилирование приложения

`perf` — инструмент ядра Linux, позволяющий отслеживать события, инициируемые исполняемой программой, записывать и анализировать их.



Профилирование приложения

`perf` — основан на анализе событий, происходящих во время работы приложения.

События могут быть программными, например отслеживающими работу с виртуальной памятью и аппаратными, например отслеживающими количество исполненных инструкций

Полный список доступных событий можно получить командой `perf list`.

Профилирование приложения

При работе с вычислительными приложениями нас в первую очередь интересуют следующие события:

- **cycles** — количество циклов процессора с начала работы приложения
- **instructions** — количество выполненных инструкций с начала работы приложения
- **branches** — переходы по ветвлениям
- **branch-misses** — ошибки в предсказании переходов
- **cache-references** — обращения в кэш
- **cache-misses** — кэш-промахи

Профилирование приложения

Зафиксировать общее количество указанных событий можно при помощи команды `perf stat`:

```
$ perf stat -e cycles,instructions,cache-misses,cache-references,branches,branch-misses ./a.out

 Performance counter stats for './a.out':

    53 318 854 915      cycles
   203 938 323 338      instructions              #    3,82  insn per cycle
         7 469 512      cache-misses              #    1,422 % of all cache refs
       525 155 014      cache-references
    17 459 844 556      branches
        59 614 988      branch-misses             #    0,34% of all branches

      17,268036743 seconds time elapsed

      17,241043000 seconds user
       0,008879000 seconds sys
```

Профилирование приложения

Обычно нас интересует не просто общая статистика исполнения, а понимание того на каких конкретно инструкциях происходили события. Записать трассы исполнения можно при помощи команды `perf record`:

```
$ perf record -e cycles,instructions,cache-misses,cache-references,branches,branch-misses ./a.out
```

А прочитать трассы можно при помощи `perf report`:

```
$ perf report
```

Для возможности сопоставлять ассемблерные инструкции с исходным кодом, профилируемое приложение следует собирать с отладочными символами (опция `-g`).

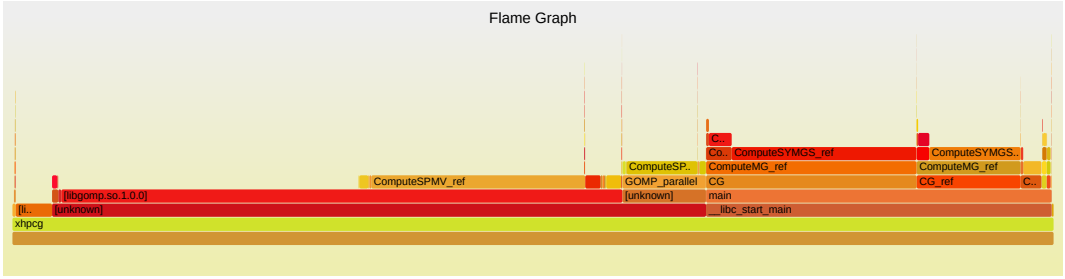
Профилирование приложения

Для сложных приложений возникает необходимость строить граф вызовов. Чтобы построить правильный граф вызовов, приложение также необходимо собирать с отладочной информацией, но при это необходимо отключить одну оптимизацию опцией `-fno-omit-frame-pointer`. Далее можно запускать `perf` с опцией `-g` (или –call-graph):

```
$ perf record -g ./a.out
```

Профилирование приложения

Помимо интерактивного режима просмотра профиля для `perf` также существует возможность визуализации графа вызовов — «пламенный граф»:



Профилирование приложения

Получение такого графа довольно простое:

```
$ git clone https://github.com/brendangregg/FlameGraph
$ perf record -g ./xhpcg 4 4 4 --rt=10
$ perf script | FlameGraph/stackcollapse-perf.pl > out.perf-folded
$ cat out.perf-folded | FlameGraph/flamegraph.pl > flame.svg
```

Заключение.

Заключение

При работе с высокопроизводительными приложениями необходимо знать и уметь пользоваться не только компилятором, но и набором прочих сопутствующих инструментов.

Попытки решать проблемы производительности без анализа узких мест обычно оборачиваются большим количеством потраченного времени и странными решениями в местах, которые скорость приложения не замедляют (по крайней мере до внесения правок).

Профилировщики позволяют быстро проанализировать и локализовать узкие места приложений и избежать выше описанных проблем.