Описание принципов работы и использования программного измерительного монитора (ПИМ) SAMPLER_v2 для измерения характеристик динамической сложности программ

1. Концепция измерения характеристик динамической сложности программ с помощью профилировщика SAMPLER_v2.

Основная особенность использования программного измерительного монитора состоит в том, что команды ПИМ для сбора измерений должна выполнять ЭВМ на которой функционирует измеряемая программа, что приводит к возникновению искажений. Количество искажений зависит от частоты обнаруживаемых событий и от операций, выполняемых монитором при обнаружении каждого события.

Наиболее распространенный тип ПИМ, к которым относится и SAMPLER_v2 - это профилировщики, также называемые анализаторами процесса выполнения программы - это программные средства, позволяющие получить ряд количественных данных о процессе выполнения исследуемой программы и на основании этих данных выявить в программе "узкие места", отрицательно сказывающиеся на эффективности ее работы. Основная цель профилировки – исследовать поведение программы во всех требуемых «точках» (машинная команда/строка исходного кода/оператор ЯВУ и т.д.), называемых контрольными точками (КТ).

Для этого профилировщики выполняют следующие функции:

- Определение общего времени выполнения каждой точки программы;
- Определение удельного времени выполнения каждой точки программы;
- Определение количества вызовов каждой точки программы;
- Определение степени покрытия программы при мониторинге.

В SAMPLER_v2 измерения выполняются в том же процессе, в котором исполняется профилируемая программа, но вся обработка снятых измерений выполняется отдельно. Это позволяет максимально упростить процедуру выполнения измерений, а также даёт возможность выполнять обработку несколько раз различными способами.

2. Методология проведения измерений с помощью ПИМ SAMPLER_v2.

Проведение измерительных экспериментов с помощью ПИМ Sampler состоит из следующих этапов:

1. Ручная расстановка контрольных точек (КТ). Для этого измеряемый код располагается между двумя функциями вызова измерителя в контрольных точках (макрос SAMPLE).
2. Компиляция исследуемой программы. При компиляции потребуется включить в программу заголовочный файла sampler.h (`#include <sampler.h>`; скорее всего, нужно будет также добавить флаг `-I` с каталогом библиотеки), а при линковке – подключить библиотеку libsampler.a (`-lsampler`; может понадобиться добавить каталог флагом `-L`). Вместе с Sampler дан пример сборки

при помощи CMake, также см. пример ниже, где для сборки Sampler используется CMake, а сборка самой профилируемой программы выполняется вручную.

3. Выполнение исследуемой программы. При использовании программы helpers/repeat, измерения выводятся в файл sampler.out.
4. Обработка снятых измерений. Для обработки используется скрипт process.py. Его работа заключается в:
   - объединении различных запусков профилируемой программы;
   - объединении различных проходов одной дуги программы;
   - вычислении среднего времени, дисперсии и СКО;
   - построении графа из набора дуг.

Подробнее см. в док-те «Концепция оценивания характеристик монитором Sampler_V2».

5. Формирование отчётов в требуемых форматах при помощи скриптов helper/fmtdot, helper/fmtshort, helper/fmttable. Fmtdot создаёт граф в формате, используемом пакетом Graphviz, что позволяет получить графическое представление модели программы. Fmttable создаёт таблицу Markdown, которую можно преобразовать в HTML или документ LibreOffice/MS Office при помощи программы Pandoc. Подробнее см. ниже.

3. Обоснование методики и точности выполнения измерений.

Для выполнения измерений используется системный вызов clock_gettime. В качестве источника времени используется CLOCK_MONOTONIC — монотонно возрастающий счётчик, не зависящий от системного времени и часовых поясов; точность 1 нс. Для данного счётчика clock_gettime на x86_64 реализуется в vDSO[1], поэтому собственно системного вызова не происходит, соответственно замеры занимают не очень много времени и не сильно влияют на кэши.

Чтобы не хранить все измерения в памяти программы, SAMPLER выводит каждое измерение в заданный файл или файловый дескриптор сразу после снятия. Таким образом, влияние на память основной программы получается небольшим (как минимум, не требуется динамически выделять память), но при сбросе буфера потока вывода может появляться довольно крупная задержка, влияние которой на проводимые измерения было бы нежелательно.

Для минимизации искажений, вызванных получением временных отметок и выводом измерений, сохраняются 4 временные отметки: две подряд в начале участка (t1 и t2), и ещё две в его конце (t3 и t4). Интервалы между двумя парами отметок используются для того, чтобы скомпенсировать (приблизительно) время выполнения clock_gettime. Таким образом, за время выполнения участка принимается

1 vDSO (virtual dynamic shared object) – виртуальная динамическая библиотека, предоставляемая ядром и содержащая реализации некоторых системных вызовов в пространстве пользователя. Позволяет избежать переключения контекста для некоторых часто используемых системных вызовов – в частности, gettimeofday и clock_gettime для некоторых счётчиков.

$(t_3 - t_2) - \frac{(t_2 - t_1) + (t_4 - t_3)}{2}$. Поскольку вывод измерения производится после снятия отметки t4 и до снятия t1 для следующего участка, время, потраченное на вывод, не учитывается.

4. Использование инструментов SAMPLER_v2

Интерфейс библиотеки

- Максос SAMPLE: контрольная точка в программе
- Функция `sampler_init(int *pargc, char **argv)`: инициализирует Sampler, используя аргументы командной строки. Убирает свои аргументы, при этом модифицируя `*pargc` и `argv`. Обработка завершается при первом необработанном аргументе или после «`--`». Обрабатываются следующие ключи:
  - `-o file`: выполнять вывод измерений в файл с данным именем;
  - `-O fd`: выполнять вывод в файловый дескриптор с данным номером;
  - `--`: завершить обработку аргументов командной строки, всё, что идёт далее, передаётся основной программе.
- Макрос SAMPLER_MACRO: если определён, макрос SAMPLER выполняет на один вызов функции меньше. Этот параметр может влиять на результаты измерений, при выполнении ЛР его использовать не рекомендуется.
- Макрос CLOCK: позволяет выбрать источник времени. Если не определён, используется CLOCK_MONOTONIC. При выполнении ЛР использовать другие значения не рекомендуется.

Программа sampler-repeat

Внешнее зацикливание и сбор вывода с программ, использующих sampler_init. Вывод производится в файл sampler.out.

Исполняемый файл программы будет создан во время сборки Sampler в каталоге сборки.

`sampler-repeat N_RUNS N_SKIP PROGRAM ARGS …`

- N_RUNS: общее количество запусков
- N_SKIP: количество запусков в начале, которые нужно пропустить. Программа всё равно запускается, но измерения не сохраняются.
- PROGRAM ARGS …: имя программы и аргументы. Перед аргументами добавляются `-O 3 --` для учитываемых запусков или `-o /dev/null` -- для не учитываемых.

Скрипт tools/process.py

Обработка сохранённого вывода Sampler-а и построение графа переходов.

Данный скрипт написан на Python 3 и требует наличия интерпретатора языка для работы. Он установлен по умолчанию в большинстве систем Linux. Проверить его

наличие можно командой `which python3` – она должна выдать путь к программе интерпретатора, если он установлен.

```
tools/process.py [INPUT [OUTPUT]]
```

- INPUT: файл с сохранённым выводом Sampler-a. По умолчанию sampler.out. Если указать «-», будет использоваться стандартный ввод.
- OUTPUT: файл, в которых производится вывод графа переходов. По умолчанию стандартный вывод («-» также обозначает стандартный вывод).

Скрипты для форматирования

Все скрипты fmt* принимают файл с обработанными результатами измерений на вход и выдают текст в заданном формате на вывод.

- fmtshort: простой текстовый вывод;
- fmtdot: вывод описания графа для Graphviz;
- fmttable: вывод Markdown таблицы.

Пример использования

Все команды начинаются с символа «$», он обозначает приглашение командной оболочки. Сам символ «$» вводить не надо. Если в начале строки не стоит этот символ, строка является продолжением предыдущей.

Выполняем сборку Sampler-a (из каталога с исходным кодом Sampler):

```
$ cmake -S . -B build -D CMAKE_BUILD_TYPE=Debug
$ cmake --build build
```

Компилируем свою программу (допустим, она называется xyz.c, а каталог с исходным кодом Sampler называется Sampler/)

```
$ gcc -o xyz xyz.c -ISampler/libsampler
-LSampler/build/libsampler -lsampler
```

Запускаем свою программу с зацикливанием и выполняем обработку:

```
$ Sampler/build/tools/sampler-repeat 10 5 ./xyz
$ Sampler/tools/process.py >xyz.json
```

Получаем из результатов PNG изображение с графом переходов (используется пакет Graphviz):

```
$ Sampler/tools/fmtdot <xyz.json | dot -Tpng -s96 >xyz.png
```

Получаем текстовую таблицу с переходами и ODT файл с таблицей для LibreOffice (используется программа Pandoc):

```
$ Sampler/tools/fmttable <xyz.json | pandoc -f markdown -t plain
$ Sampler/tools/fmttable <xyz.json | pandoc -f markdown -t odt
>xyz.odt
```

Вместо раздельных шагов обработки и генерации представления, иногда (например, при подборе параметров зацикливания) можно использовать команду, выполняющую обработку и вывод табличного представления в терминал:

```
$ Sampler/tools/process.py | Sampler/tools/fmttable | pandoc -f
markdown -t plain
```

5. Представление и использование результатов измерений.

Пример отчета по результатам измерений, сформированного в виде таблицы для программы test_sub.c:

```
 1 #include "sampler.h"
 2
 3 #ifndef SIZE
 4 #define SIZE 10000
 5 #endif
 6
 7 void TestLoop(int nTimes)
 8 {
 9     static int TestDim[SIZE];
10     int tmp;
…
30     SAMPLE;
31     TestLoop(SIZE / 10);
32     SAMPLE;
33     TestLoop(SIZE / 5);
34     SAMPLE;
35     TestLoop(SIZE / 2);
36     SAMPLE;
37     TestLoop(SIZE / 1);
38     SAMPLE;
39 }
```

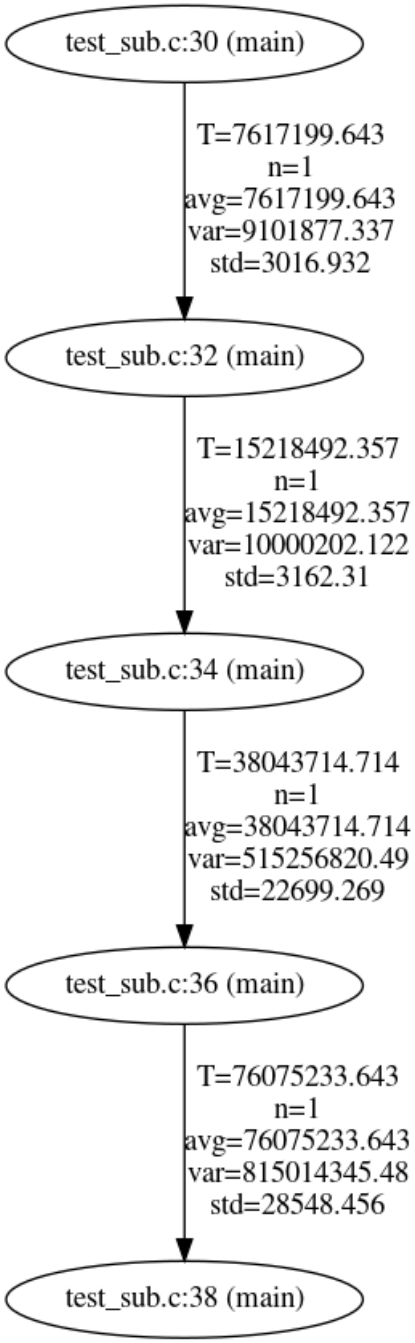
Пример таблицы

Числа в колонках «исх» и «прием» соответствуют номерам в строк в таблице выше.

исх	прием	общее время	кол-во проходов	среднее время
30	32	7617199.643	1	7617199.643
32	34	15218492.357	1	15218492.357
34	36	38043714.714	1	38043714.714
36	38	76075233.643	1	76075233.643

Пример графа переходов

Пример отчета по результатам измерений, сформированного в виде графа тестовой программы имеет вид:



Из приведенного примера видно, что вершины управляющего графа программы (УГП) соответствуют местам вызова функций КТ, а нагрузка дуг задает времена и количество выполнений соответствующих участков программы. Кроме того, на подписях к дугам указаны число переходов (n), среднее время выполнения дуги (avg), а также дисперсия (var) и СКО (std) времени выполнения.

6. Тестовые программы: test_cyc и test_sub

При выполнении ЛР предлагается выполнить профилирование двух тестовых программ: test_cyc и test_sub:

- Программа test_cyc содержит несколько циклов, выполняющих перестановку элементов одного статического массива. Первый цикл обрабатывает первую 1/10 элементов, следующий 2/10, третий 5/10, а четвёртый обрабатывает массив целиком; затем такая же последовательность повторяется ещё дважды. Несмотря

на то, что циклы выполняют одни и те же действия, их производительность различается из-за того, что первые 4 цикла обеспечивают попадание массива в кэш процессора. По умолчанию в программе используется массив `int[10000];` при размере int 4 байта это чуть менее 40 килобайт. Передачей другого размера массива в макросе SIZE можно получить массив, который перестанет влезать в кэш данных первого уровня, а затем в кэши второго и третьего уровней (посмотреть размеры кэшей в системах Linux можно при помощи команды lscpu). В этом случае пятый и далее циклы не будут получать ускорения.

- Программа test_sub содержит такой же цикл, как и test_cyc, но вынесенный в функцию. По этой программе можно проверить, что время исполнения цикла в функции не будет заметно отличаться от времени исполнения цикла вне её при том же объёме работы.