ИЗМЕРЕНИЕ ПОКРЫТИЯ КОДА ПРИ РАБОТЕ БЕЗ ОПЕРАЦИОННОЙ СИСТЕМЫ ВСТРОЕННЫМИ СРЕДСТВАМИ КОМПИЛЯТОРА GCC

Андрианов А.В.

andrianov@module.ru

ЗАО НТЦ “Модуль”

В статье рассматривается проблема измерения тестового покрытия кода при работе без операционной системы с использованием инструментария GNU C Compiler (gcc). В частности, изучается задача получения тестового покрытия кода, работающего на RTL модели СБИС. Приводятся примеры реализации функций, необходимых для получения тестового покрытия, и шаги, необходимые для дальнейшего анализа и создания отчетов по тестовому покрытию.

Введение

Сложность современных задач цифровой обработки сигналов требует привлечения все более сложных систем тестирования и визуального отображения и анализа информации о качестве тестов. Это требуется как на этапе разработки и отладки библиотек алгоритмов, так и на стадии проектирования СБИС.

Тестовое покрытие – это один из наиболее часто применяемых инструментов, которое позволяет узнать, какие именно участки кода исполнялись в ходе тестирования, какие именно ветви программы были исполнены, и, соответственно, сделать заключение о полноте выполненного тестирования. Для получения тестового покрытия требуется поддержка данной функции компилятором. При включении данной функции, компилятор дополняет (инструментирует) генерируемый код дополнительными инструкциями, ведущими в ходе выполнения учет вызова функций и переходов. После выполнения программы, данная информация записывается в файл для дальнейшего анализа и генерации отчетов.

В данной статье будет рассматриваться инструментальная цепочка GNU (gcc) и связанный с ней набор инструментов для получения и анализа тестового покрытия. Шаги, необходимые для получения тестового покрытия программы, компилируемым другим компилятором, могут существенно отличаться.

Обзор последовательности сбора и анализа покрытия

Для начала рассмотрим процесс компиляции программы. При компиляции исходного кода на языках C/C++ необходимо передавать компилятору опции *-fprofile-arcs* и *-ftest-coverage*. Эти опции включают инструментирование кода счетчиками, в которых будут регистрироваться переходы и вызовы функций в ходе выполнения программы. На стадии компоновки – необходимо задействовать опцию *--coverage*. Также, при работе под управлением ОС необходима компоновка кода с библиотекой gcov (при помощи опции *-lgcov*).

Полный список команд, необходимых для сборки программы из трех файлов (1.c, 2.c и 3.cpp соответственно), представлен на Листинге 1. Файл 3.cpp не инструментируется.

```
gcc -fprofile-arcs -ftest-coverage -c -o 1.o 1.c
gcc -fprofile-arcs -ftest-coverage -c -o 2.o 2.c
g++ -c -o 3.o 3.cpp
g++ 1.o 2.o 3.o -lgcov --coverage
```

Листинг 1. Последовательность компиляции программы с инструментированием для

снятия покрытия

Большое количество функций, покрытие которых изучается, может существенно увеличить время моделирования. Поэтому отдельные файлы можно исключать из покрытия компилируя их без данных опций. Например, так можно поступать с файлами, содержащими исходный код запускаемых тестовых сценариев и не содержащих исходного кода библиотечных функций, для которых требуется точное покрытие.

При сборке программы, которая была инструментирована для записи покрытия, помимо стандартных объектных (*.o) файлов будут сгенерированы файлы аннотации с расширением gcno, которые потребуются в дальнейшем. При запуске программы будут созданы файлы gcda, содержащие информацию о выполненных функциях и переходах. При этом существующие gcda файлы дополняются вновь собранными данными. Таким образом, можно запустить один и тот же тест несколько раз, с разными параметрами и получить агрегированную статистику.

В дальнейшем, для анализа используются следующие инструменты:

- gcov – позволяет выводить разнообразную информацию о покрытии в текстовом формате;
- lcov/genhtml – служит для совмещения собранного покрытия в общий файл и при помощи вспомогательного инструмента genhtml может сгенерировать интерактивный html файл, содержащий информацию о покрытии

```
# Сборка и компоновка программы
gcc -fprofile-arcs -ftest-coverage -c -o 1.o 1.c
gcc -fprofile-arcs -ftest-coverage -c -o 2.o 2.c
g++ -c -o 3.o 3.cpp
g++ 1.o 2.o 3.o -lgcov --coverage
# Генерация .info файла при помощи lcov
lcov --directory . -c -o test.info
# Объединение нескольких отчетов lcov в один
lcov -a test.info -a test2.info -o full.info
# Генерация интерактивного html отчета
genhtml –o html/ full.info
```

Листинг 1. Последовательность компиляции программы с инструментированием для снятия покрытия и генерация интерактивного отчета

Особенности сбора покрытия при работе без операционной системы

При работе без ОС в большинстве случаев файловые операции недоступны, а в используемой инструментальной цепочке будет отсутствовать библиотека gcov. Это также касается случаев, когда покрытие собирается для компонентов ядра ОС. Для того, чтобы обеспечить сбор тестового покрытия при работе программы без ОС необходимо:

- Добавить поддержку конструкторов и (опционально) деструкторов начальным кодом инициализации платформы (если она отсутствует) и/или используемым ld сценарием.
- Реализовать возможность передать сгенерированные gcda данные на хост-компьютер и записать их в файлы в директории сборки, в директорию с gcno файлами
- Реализовать функции __gcov_init и __gcov_exit

- Убедиться, что в ld сценарии присутствует секция .data

Дальше эти пункты будут рассмотрены более детально.

Поддержка конструкторов в gcc

Конструкторы и деструкторы в gcc объявляются используя атрибуты функции constructor и destructor соответственно. Пример приведен на Листинге 2.

```
__attribute__((constructor)) void my_init_func() {
  dosomething();
}

__attribute__((destructor)) void my_deinit_func() {
  dosomethingelse();
}
```

Листинг 2. Пример определения конструкторов и деструкторов в gcc

При компоновке, указатели на конструкторы помещаются в секции .pre_init_array и .init_array соответственно, а адреса начала и конца массива указателей должны определяться сценарием компоновки (См. Листинг 3). Данные функции должны вызываться специфичным для платформы кодом инициализации перед выполнением функции main.

```
    .initfini:
    {
      . = ALIGN(4);
       /* preinit data */
       PROVIDE_HIDDEN (__preinit_array_start = .);
       KEEP(*(.preinit_array))
       PROVIDE_HIDDEN (__preinit_array_end = .);

       . = ALIGN(4);
       /* init data */
       PROVIDE_HIDDEN (__init_array_start = .);
       KEEP(*(SORT(.init_array.*)))
       KEEP(*(.init_array))
       PROVIDE_HIDDEN (__init_array_end = .);


       . = ALIGN(4);
       /* finit data */
       PROVIDE_HIDDEN (__fini_array_start = .);
       KEEP(*(SORT(.fini_array.*)))
       KEEP(*(.fini_array))
       PROVIDE_HIDDEN (__fini_array_end = .);
}
```

Листинг 3. Фрагмент ld сценария для поддержки конструкторов/деструкторов

В случае доступности стандартной библиотеки языка C (например, newlib) цепочку конструкторов и деструкторов можно запустить, вызвав функции __libc_init_array и __libc_fini_array

соответственно. Пример реализации данных функций можно увидеть в исходных кодах библиотеки newlib ([2] и [3]).

Реализация функции __gcov_init

При задействованном инструментировании кода компилятор поместит в секции __gcov0.* структуры типа struct gcov_info (по одной на каждый объект компиляции). Они статически инициализированы на этапе компиляции и должны загружаться в оперативную память. В программу так же будут добавлены специальные конструкторы, которые будут вызывать функцию __gcov_init на каждую из структур __gcov_info. Пример реализации функции __gcov_init можно увидеть на Листинге 4. В минимальной реализации достаточно просто объединить все имеющиеся элементы в один связанный список и сохранить указатель на первый элемент из списка.

```
struct gcov_info {
        unsigned int version;
        struct gcov_info *next;
        unsigned int stamp;
        const char *filename;
        void (*merge[GCOV_COUNTERS])(gcov_type *, unsigned int);
        unsigned int n_functions;
        struct gcov_fn_info **functions;
};

static struct gcov_info *gcov_info_head;

void gcov_info_link(struct gcov_info *info)
{
        info->next = gcov_info_head;
        gcov_info_head = info;
}

void __gcov_init(struct gcov_info *info)
{
        gcov_info_link(info);
}
```

Листинг 4. Вариант реализации __gcov_init

Формат структур для учета покрытия может меняться в зависимости от версии gcc. На момент написания данной статьи существует два возможных формата: используемый в gcc, начиная с версии 3.4 и более новый, используемый в gcc начиная с версии 4.7. Готовая реализация функций для работы с этими форматами есть в коде соответствующего модуля ядра linux[1] и исходном коде компилятора gcc[4].

Так же сгенерированный gcc код может вызывать другие функции, реализацию которых можно оставить пустой. Полный их список приведен на Листинге 5.

```
void __gcov_flush(void);
void __gcov_merge_add(gcov_type *counters, unsigned int n_counters);
```

```
void __gcov_merge_single(gcov_type *counters, unsigned int n_counters);
void __gcov_merge_delta(gcov_type *counters, unsigned int n_counters);
void __gcov_merge_ior(gcov_type *counters, unsigned int n_counters);
void __gcov_merge_time_profile(gcov_type *counters, unsigned int n_counters);
void __gcov_merge_icall_topn(gcov_type *counters, unsigned int n_counters);
void __gcov_exit(void);
```

Листинг 5. Функции-заглушки, которые может вызывать код, сгенерированный gcov

Генерация данных в формате gcda

После исполнения тестового сценария перед завершением программы необходимо преобразовать данные, хранящиеся в структурах struct gcov_info в формат gcda и сохранить в виде файлов на хост-системе.

Структура gcov_info содержит поле filename, в которое при компиляции записывается путь к файлу gcda, в который необходимо записать данные. Пример функции, выполняющей обход связанного списка, преобразование в gcda и сохранение всех gcov_info объектов, приведен на Листинге 6.

```
void save_coverage()
{
        printf("gcov: Dumping coverage...\n");
        struct gcov_info *info = gcov_info_next(NULL);
        char *tmp;
        uint32_t len;
        while(info) {
          len = gcov_convert_to_gcda(NULL, info);
          tmp = malloc(len);
          gcov_convert_to_gcda(tmp, info);
          const char *file = gcov_info_filename(info);
          platform_dump_region(file, (uint32_t) tmp, len);
          info = gcov_info_next(info);
          free(tmp);
        }
#endif
}
```

Листинг 6. Сохранение gcda

Реализация функции gcov_convert_to_gcda превышает допустимый объем данной статьи, потому не была включена в листинг. Ее реализацию можно посмотреть в исходном коде ядра linux[1].

Функция platform_dump_region() должна сохранять содержимое переданного буфера в файл на хост системе. Ее реализация зависит от окружения. В случае работы на модели, это может быть инструкция для верификационного окружения, которая произведет сохранение данных в файл. В случае работы на реальной аппаратуре – передача данных по последовательному порту, вызов RPC процедуры, или запись на ПЗУ.

После сохранения .gcda файлов работать с покрытием и генерировать отчеты можно при помощи стандартных инструментов.

Параллельный запуск тестов с генерацией покрытия

Реализация сохранения gcda данных, приведенная выше, не учитывает того, что gcda файл может уже существовать. Загрузка существующих gcda файлов и объединение при запуске очередного теста потребует намного более сложной реализации, которая будет существенно усложнять работу на модели СБИС.

В большинстве случаев, одна и та же библиотека покрыта целым набором тестов, которые запускаются независимо. Это будет приводить к тому, что gcda файлы будут перезаписываться, и часть информации о покрытии теряться. Самым простым решением данной проблемы может стать объединение всех тестов в одну программу. Но это может быть невозможно из-за ограниченного объема памяти или большого времени последовательного исполнения тестов (особенно при моделировании СБИС).

Более простым решением здесь может служить сохранение gcda файлов не по указанному в gcov_info абсолютному пути, а в пути относительно рабочей директории, относящейся к данному тесту. При этом необходимо скопировать (или создать символические ссылки) файлы аннотации (.gcno) и разместить их рядом. Это позволит запускать несколько тестов параллельно. После завершения запуска всех тестов, будет необходимо сгенерировать при помощи lcov .info файлы на каждый из тестов, после чего объединить их при помощи lcov, и только после этого сгенерировать html документацию из .info файла, содержащего информацию обо всех тестах. (См. Листинг 1).

Заключение

Тестовое покрытие может дать важную информацию о том, насколько полно выполнена верификация программного кода, а также указать ветви, которые необходимо проверить дополнительными тестами. Использование встроенных средств компилятора позволяет максимально избежать внесения в код изменений, связанных со сбором и анализом данных, а также связанных с этим ошибок.

Литература

1. https://elixir.free-electrons.com/linux/v4.15-rc9/source/kernel/gcov
2. https://github.com/eblot/newlib/blob/master/newlib/libc/misc/init.c
3. https://github.com/eblot/newlib/blob/master/newlib/libc/misc/fini.c
4. https://github.com/gcc-mirror/gcc/blob/master/libgcc/libgcov-driver.c
5. Robert L. Glass, "Persistent Software Errors," IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, March, 1981.
6. Muhammad Shahid, Suhaimi Ibrahim, An Evaluation of Test Coverage Tools in Software Testing, 2011 International Conference on Telecommunication Technology and Applications, Proc .of CSIT vol.5 (2011) © (2011) IACSIT Press, Singapore

This article describes test coverage generation when working without any operating system using the GNU C Compiler (gcc) toolchain. Generating test coverage for code running on an RTL model of a “System-on-Chip” (SoC) is also covered. The article

contains a sample implementation of functions required to obtain test coverage and all the steps required for further analysis and coverage report creation.