Методы профилирования и оптимизации кода для процессоров семейства Blackfin

Павел ФРОЛОВ
Игорь ЛЕКТОРОВ
info@promwad.com

Оптимизация кода в процессе разработки программного обеспечения для использования во встроенной системе играет если не первостепенную, то, по крайней мере, значимую роль. Средства для разработки ПО не могут обеспечить полноценное создание кода и использование полного функционала появляющихся новых процессоров.

Введение

При усложнении процессорных архитектур становится все труднее использовать язык assembler, все чаще прибегают к абстракции через использование языка C либо операционных систем. Это, с одной стороны, позволяет быстро выйти на рынок с новой архитектурой и стандартными средствами разработки GCC, а с другой — не позволяет достичь расчетной максимальной производительности для данной архитектуры.

Во многом это явление характерно для архитектуры Blackfin: в погоне за функционалом, предоставляемым операционной системой Linux, порой забывают о той способности процессоров цифровой обработки сигналов, ради которой они изначально создавались. Чтобы получить более эффективный код на выходе, разработчики часто прибегают к переписыванию частей кода на ассемблере. Однако это не является единственным способом повышения производительности.

Во-первых, можно использовать специфические встроенные функции (***built-in functions***), которые написаны с учетом данной архитектуры.

Во-вторых, доступны для использования библиотеки.

Инженер компании Analog Devices Робин Гетс (Robin Getz)утверждает, что использование библиотек (специфических для обработки сигналов) и встроенных функций компилятора позволяет улучшить производительность кода без необходимости изучения всей особенности архитектуры процессоров Blackfin (рис. 1).

Рис. 1. Отладочная плата ADSP-BF537 EZ-KIT LITE

Основные термины и определения

Профилирование — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов, число верно предсказанных условных переходов, число кэш-промахов и т. д.

Профилировщик — инструмент, используемый для анализа работы программы.

uClinux — Linux-подобная встраиваемая операционная система для микроконтроллеров, не имеющих блока управления памятью. Начиная с версии ядра 2.5.46 проект был объединен с основной линией разработки ядра Linux. Он распространяется в виде дистрибутива ***uClinux-dist*** и может использоваться на встраиваемых устройствах. ***uClinux-dist*** содержит программные библиотеки, приложения и утилиты. Его можно сконфигурировать и встроить в ядро системы.

GCC (GNU Compiler Collection) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU.

Blackfin Toolchain (кросс-компилятор) — набор необходимых пакетов программ для компиляции и генерации исполняемого кода из исходных текстов программ для архитектуры Blackfin.

Термин «профилирование» впервые был представлен в 1971 г. в одной из работ Дона Кнута (Don Knut). С помощью профилирования определяются участки программы, поглощающие бóльшую часть времени работы, и в них вносятся улучшения. Инструментом профилирования является программа-профилировщик, которую и используют для анализа программы.

Опции, управляющие оптимизацией GCC

Для кросс-компилятора bfin-linux-uclibc-gcc, как и для компиляторов других архитектур, существует набор флагов, информирующих компилятор о необходимости применения оптимизации:

- ***-O0*** — не использовать оптимизацию.
- ***-O***, ***-O1*** — компилятор пытается уменьшить размер кода без оптимизации, для которой требуется большее время компиляции.
- ***-O2*** — выполняются почти все поддерживаемые оптимизации, которые не включают уменьшение времени исполнения за счет увеличения длины кода.
- ***-O3*** — включает все оптимизации, определяемые ***-O2***, добавляя разворачивание циклов и выполняя функции inline, что приводит не только к увеличению размера, но и к увеличению производительности.
- ***-Os*** — оптимизация по размеру исполняемого кода.

Также можно использовать некоторые флаги вида ***-fфлаг***, например, ***-ffast-math*** (не рекомендуют использовать с одним из ***O#***), ***-fomit-frame-pointer***.

В пакете uClinux-dist для измерения производительности есть программа ***whetstone***, состоящая из большого количества операций с плавающей точкой. Более 50% вре-

Таблица 1. Результаты измерений

Флаги кросс-компилятора	Размер файла, байт	Время выполнения, с
-O0	38 540	321
-O1	27 452	253
-O2	27 324	74
-O3	28 916	65
-Os	27 052	73
-O0 -mfast-fp	31 492	228
-O1 -ffast-math -mfast-fp	17 380	159
-O2 -mfast-fp	20 276	57
-O3 -ffast-math	25 800	26
-O3 -ffast-math -mfast-fp	18 788	18

мени программа выполняет математические операции с плавающей точкой. Эта программа собиралась с различными опциями оптимизации и запускалась на процессоре ADSP-BF537 (частота процессора — 500 МГц, частота шины памяти — 125 МГц) с количеством циклов 30 000. Результаты некоторых измерений приведены в таблице 1. Полную таблицу для этого теста можно найти в [1].

Применение встроенных функций

Для эффективного использования архитектурных особенностей процессора Blackfin используют встроенные функции (например, ***built-in function***).

Существуют операции как с 16-битными, так и с 32-битными числами с плавающей и фиксированной точкой, а также операции перестановки байтов в слове и операции умножения младших частей слова на старшие.

К сожалению, не реализована возможность использования циклических буферов и запись в соответствии с обратным двоичным порядком, что специфично для БПФ.

Некоторые операции для работы с 16-битными числами представлены в таблице 2 (для функций используется префикс ***__builtin_bfin_***). Полный список встроенных функций можно найти в [2].

Таблица 2. Операции для работы с 16-битными числами

Функция	Аргументы	Операция
add_fr1x16	(fract16 f1, fract16 f2)	Сложение 16-битных чисел
sub_fr1x16	(fract16 f1, fract16 f2)	Разность 16-битных чисел
min_fr1x16	(fract16 f1, fract16 f2)	Возвращает минимальное значение из двух чисел
shl_fr1x16	(fract16 src, short shft)	Арифметический сдвиг влево переменной src на shft разрядов

Использование кэша L1

Самой быстрой памятью процессора является кэш первого уровня — L1-cache. По сути, он является неотъемлемой частью процессора, поскольку расположен на одном с ним кристалле и входит в состав функциональных блоков. Память L1 работает на частоте процессора.

Суммарный размер памяти L1 для процессора ADSP-BF537 составляет 132 кбайт, из которых:

- 64 кбайт являются статической памятью инструкций.
- 64 кбайт являются статической памятью данных.
- 4 кбайт являются сверхоперативным буфером для данных и не могут быть сконфигурированы как кэш.

При конфигурировании памяти L1 как кэш будет доступно следующее пространство:

- 16 кбайт — для кэша инструкций;
- 32 кбайт — для кэша данных.

Размещение исполняемого кода (программ, данных, функций) в кэше L1 позволяет повысить производительность за счет большей скорости доступа к памяти.

Учитывая ограниченный размер памяти L1 и, как правило, значительно бóльший размер исполняемого файла, наиболее полезным является размещение часто используемых и ресурсоемких функций в L1.

Чтобы включить поддержку размещения исполняемого кода в кэше L1, необходимо включить следующие опции процессора в ядре Linux: Enable ICACHE и Enable DCACHE.

```
Blackfin Processor Options
--->
    --- Cache Support
    [*] Enable ICACHE
    [*] Enable DCACHE
     [ ] Enable Cache Locking
         Policy (Write back)
    --->
```

Для размещения приложения в L1 необходимо использовать FDPIC-формат исполняемых файлов и выполнить сборку с опциями компилятору (CFLAGS):

```
-fno-jump-tables
```

и линкеру (LDFLAGS):

```
-pie -Wl,--sep-code -Wl,--code-in-l1 -Wl,-z,now
```

Размещение приложения в L1:

```
CFLAGS += -fno-jump-tables
LDFLAGS += -pie -Wl,--sep-code -Wl,--code-in-l1 -Wl,-z,now
```

Размещение данных в L1:

```
CFLAGS += -Wl,--data-in-l1
```

Размещение отдельных функций и переменных в L1:

```
void foo() __attribute__ ((l1_text))
int var __attribute__ ((l1_data_A))
CFLAGS += -fno-jump-tables
```

Динамическое выделение памяти в SRAM:

```
void *sram_alloc (size_t size, unsigned long flags)
int sram_free (void *addr)
void *dma_memcpy (void *dest, const void *src, size_t size)
```

Более подробная информация по использованию кэша L1 находится в [3].

Использование библиотеки libbfdsp

Библиотека libbfdsp представляет собой частично портированную с VisualDSP++ библиотеку, содержащую набор функций для цифровой обработки сигналов. Предоставляются

Таблица 3. Комплексные функции

Операция	Прототип функции
Сложение	complex_double cadd (complex_double a, complex_double b)
Вычитание	complex_double csub (complex_double a, complex_double b)
Умножение	complex_double cmlt (complex_double a, complex_double b)
Деление	complex_double cdiv (complex_double a, complex_double b)

Таблица 4. Фильтры

Операция	Прототип функции
КИХ	void fir_fr16 (const fract16 input[], fract16 output[], int length, fir_state_fr16 *filter_state)
БИХ	void iir_fr16 (const fract16 input[], fract16 output[], int length, iir_state_fr16 *filter_state)
Прямая форма БИХ	void iirdf1_fr16 (const fract16 input[], fract16 output[], int length, iirdf1_fr16_state *filter_state)

такие функции, как БПФ, операции свертки, комплексные умножения и сложения векторов, расчет фильтров с конечной и бесконечной импульсной характеристиками (табл. 3, 4).

Поскольку Blackfin не содержит сопроцессор плавающей точки, ***GCC*** предоставляет библиотеку эмуляции операций с плавающей точкой ***libbffastfp***, оптимизированную для данной архитектуры и написанную на ассемблере. Для ее использования добавляется флаг ***-mfast-fp***. Однако следует заметить, что исполнение этой библиотеки не соответствует некоторым правилам IEEE-стандарта для чисел с плавающей точкой (например, нет проверки на NaN) для большей производительности [4, 5].

Измерение производительности и профилирование

Прежде чем начать профилирование на встроенной архитектуре, часто бывает полезно изучить распределение производительности на персональном компьютере с помощью утилиты ***Valgrind***.

Valgrind — профилировщик для PC, имеющий общую инфраструктуру и позволяющий использовать различные инструменты профилирования.

Для этого пользовательское приложение собирается без оптимизации и с отладочной информацией (флаги ***gcc -O0 -g***), после чего запускается командой:

```
valgrind --tool=callgrind whetstone
```

После завершения выполнения приложения в текущем каталоге создается файл с информацией профилировщика — ***callgrind.out.#***, который может быть обработан утилитой ***KCachegrind*** (графический интерфейс KDE) с получением удобочитаемого графика вызовов функций.

В результате анализа графика вызова функций можно определить наиболее часто вы-

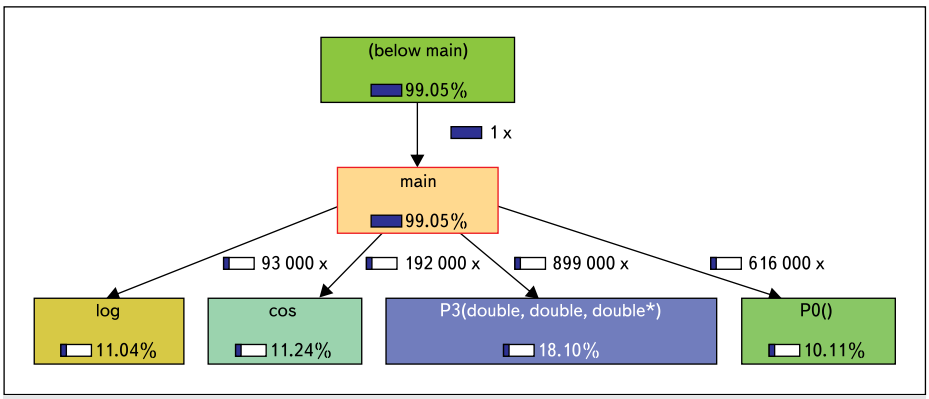

Рис. 2. Часть дерева вызовов утилиты whetstone

зываемые функции, чтобы попытаться их оптимизировать в дальнейшем. Проблема использования ***Valgrind*** для встроенных систем состоит в том, что данная утилита для них недоступна, и результат ее работы на PC может значительно отличаться от реальности.

На рис. 2 представлена часть дерева вызовов утилиты ***whetstone***.

Простейший способ измерения производительности для процессоров архитектуры Blackfin

Стандартный способ построения утилит для профилирования состоит в том, что при входе в функцию и при выходе из нее высчитывается время (либо счетчик тактов) и берется их разность.

Для процессоров Blackfin можно считывать регистры ***cycles*** и ***cycles2***, чтобы определить количество тактов с момента сброса процессора. Функция, возвращающая необходимое значение тактов, имеет вид:

```
static inline unsigned long long read_cycles (void)
{
        unsigned long long t0;
        asm volatile ("%0=cycles; %H0=cycles2;" : "=d" (t0));
        return t0;
}
```

В программе для измерения количества тактов процессора, затраченных для выполнения функции, делают следующее:

```
unsigned long start, end;
user_function();              //загрузка функции в кэш;
start = read_cycles();        //тактов перед измеряемой функцией;
user_function();              //функция для измерений;
end = read_cycles;            //тактов после измеряемой функции;
printf("Results: %lu sec\n",  (end-start)/cpufrequency);
```

Утилита GCOV

Довольно удобным средством профилирования, как для PC, так и для встроенных систем, является утилита ***GCOV*** (***GNU Coverage***), входящая в комплект ***GCC***. ***GCOV*** позволяет на основе имеющихся исходных кодов создавать аннотированные исходные коды.

Для ее применения при сборке приложения задаются опции компилятора, которые добавляют метки входа и выхода во все функции:

```
CFLAGS += -fprofile-arcs -ftest-coverage -g -O0
```

При этом создаются файлы с расширением ****.gcno***.

Далее полученное приложение запускается на целевой архитектуре (в нашем случае — на процессоре Blackfin). После завершения программы генерируются файлы ****.gcda*** (по умолчанию, если не указана переменная окружения ***GCOV_PREFIX***, данные файлы создаются по тому же пути, где находятся исходные коды программы). Остается только скачать все файлы ****.gcda*** в каталог с исходными кодами и запустить утилиту ***gcov***:

```
bfin-linux-uclibc-gcc file.gcda
```

В результате получаем аннотированные файлы ****.gcov***, по которым можно определить наиболее затратные по производительности места программы.

Первое поле показывает количество вызовов данного участка, второе — номер строки.

Также возможно применять ***ggcov*** — утилиту с графическим интерфейсом, показывающую покрытие кода, количество входов в ту или иную функцию.

Участок примера аннотированного исходного кода представлен в таблице 5.

Таблица 5. Участок примера аннотированного исходного кода

Количество входов	Номер строки	Исходный код
–	267	
32 001	268	for (I = 1; I <= N7; I++)
–	269	{
32 000	270	X = T * DATAN(T2*DSIN(X)*DCOS(X));
32 000	271	Y = T * DATAN(T2*DSIN(Y)*DCOS(Y));
–	272	}

Пакет Oprofile

Хорошей альтернативой для встроенного профилировщика является пакет ***Oprofile***, который состоит из двух частей: уровня ядра и уровня пользователя. То есть при конфигурировании ядра и пользовательских программ, входящих в ***uClinux-distribution***, должны быть выбраны соответствующие опции. Единственным требованием для приложений, собранных для профилирования, является параметр ***-g***, для включения отладочной информации.

Пакет ***Oprofile*** состоит из нескольких утилит:

- ***bfin_opcontrol*** — управляющий скрипт;
- ***oprofiled*** — домен, собирающий информацию о профилировании;
- ***opreport*** — скрипт, выводящий отчет о профилировании.

Для запуска ***Oprofile*** необходимо выполнить следующие операции:

- Проинициализировать ***Oprofile***:

```
bfin_opcontrol –init
```

- Запустить домен ***Oprofile***, указав ему исполняемый файл в опции ***–image***:

```
oprofiled -e ' ' --no-vmlinux --image=/bin/whetstone &
```

- Выполнить старт ***Oprofile***:

```
bfin_opcontrol --start
```

Далее запустить приложение и в процессе его выполнения (или по завершению) отслеживать распределение ресурсов.

Вывод данных профилирования:

```
bfin_opcontrol --dump
opreport -l
```

Приведем для примера часть отчета, предоставляемого утилитой ***opreport***::

```
Profiling
through timer
interrupt
    samples % symbol name
        1128 31.9005 ___divdf3
        580 16.4027 ___unpack_d
        454 12.8394 ___muldf3
        403 11.3971 __fpadd_parts
        376 10.6335 ___pack_d
        356 10.0679 ___muldi3
        122 3.4502 ___adddf3
        44 1.2443 _P3
        30 0.8484 _P0
        20 0.5656 ___subdf3
        10 0.2828 _PA
```

Видно, что бóльшую часть процессорного времени занимают операции деления, распаковки и умножения с плавающей точкой.

Если выполняемое приложение использует разделяемые библиотеки, то в выводе ***Oprofile*** будет отображаться количество ресурсов, занимаемое той или иной библио-

текой. В качестве примера приведем кодирование оцифрованного звука формата ***wav*** в формат ***ogg***:

```
Profiling
through timer
interrupt
    samples % app name
        143251 76.2448 libavcodec.so.51.48.0
        28086 14.9487 no-vmlinux
        10520 5.5992 libavformat.so.52.1.0
        3866 2.0577 libgcc_s.so.1
        664 0.3534 libuClibc-0.9.29.so
        508 0.2704 libavutil.so.49.5.0
        336 0.1788 libuClibc-0.9.29.so
```

Разделяемые библиотеки можно отдельно профилировать с помощью ***Oprofile***, указав их название в опции ***--image***.

Заключение

Знание и учет архитектурных особенностей процессоров Blackfin, использование их мультимедийных возможностей при разработке ПО позволяет создавать высокопроизводительные приложения для встраиваемых архитектур. Использование же свободной операционной системы Linux (как на PC, так и во встраиваемой системе), а именно существующих специфических утилит отладки и профилирования, постоянно развивающихся и дополняющихся новым функционалом, имеющих порты на другие (отличные от обычного PC) архитектуры, делает процесс разработки и отладки ПО для встраиваемой системы более быстрым и легким.

Статья подготовлена на основе материалов форума разработчиков цифровой электроники, организованного дизайн-центром электроники Promwad.

Литература

1. **http://docs.blackfin.uclinux.org/doku.php?id=uclinux-dist: whetstone**
2. **http://docs.blackfin.uclinux.org/doku.php?id=toolchain: built-in_functions**
3. **http://docs.blackfin.uclinux.org/doku.php?id=linux-kernel: on-chip_sram**
4. **http://docs.blackfin.uclinux.org/doku.php?id=toolchain: libbfdsp**
5. **http://www.analog.com/static/imported-files/software_manuals/50_blackfin_cc.rev5.1.pdf**