Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Профилирование операционных систем реального времени

Курсовая работа студента 344 группы

Дерюгина Дениса Евгеньевича

Научный руководитель . . . . . асп. каф. систем. прогр. Козлов Антон Павлович

Санкт-Петербург

2014

Содержание

Введение

Профилирование - это сбор различных характеристик работы программы, таких как, например, время выполнения определённых участков кода, объём занимаемой программой памяти, эффективность алгоритмов предугадывания условных переходов. Использование профилировщиков (инструментов профилирования) важно во многих областях: разработчики оборудования могут узнавать, как ведёт себя старое ПО на новой платформе, разработчики ПО могут находить те самые 5% кода, которые исполняются 95% времени, или, например, проверить, насколько эффективна стратегия замещения данных в кэш-памяти, при реинжиниринге можно производить анализ покрытия кода для удаления неиспользуемых участков.

По мере роста объёма программы всё сложнее становится понимать поведение как отдельных её модулей, так и поведение системы в целом. При отсутствии соответствующих инструментов тестирование, отладка и оптимизация становятся в разы сложнее.

Понимание поведения программ становится особенно важными при изучении работы операционных систем реального времени, так как для таких ОС важна детерминированность (то есть определённость в поведении), чего сложно достигнуть без соответствующих инструментов. Более того, в таких системах важно собирать информацию не только о работе прикладного ПО как такового, но и о работе ядра.

Практическая часть моей курсовой работы основана на модульной конфигурируемой операционной системе Embox для встроенных систем, так как этот проект имеет открытый исходный код, а также имеется возможность непосредственно контактировать с разработчиками.

Проект Embox существует уже несколько лет, за это время было написано более 150 000 строк кода (преимущественно на языке C), реализовано множество алгоритмов от классики для операционных систем, описанной в литературе, до оригинальных разработок. Имеется поддержка ряда платформ (ARM, x86, MicroBlaze, MIPS, PowerPC, SPARC).

1 Обзор методов профилирования

Профилировщики не являются новшеством - подобные инструменты существовали по крайней мере на IBM/360 и первых UNIX-системах. Уже тогда большую часть профилировщиков можно было отнести к одной из двух категорий: сэмплирующие (статистические) и инструментирующие, - эта же классификация актуальна до сих пор.

1.1 Сэмплирование

Идея сэмплирования заключается в том, чтобы с некоторой периодичностью приостанавливать работу программы тем или иным образом, затем производить анализ состояния программы (стэк вызовов, объём занимаемой памяти и так далее), а затем на основе собранной информации строить статистическую модель.

Профилировщики данного типа хорошо подходят для программ уровня пользователя. Вот некоторые из них, получившие распространение: Intel VTune Amplifier, AMD CodeAnalyst, Apple Inc. Shark, OProfile.

Этот подход хорош тем, что профилируемый код сам по себе не изменяется. Также относительно мало количество накладных расходов.

Сэмплирующие профилировщики могут быть основаны на двух механизмах:

1. **Программное сэмплирование** - производится с помощью системных прерываний. Большинство профилировщиков использует именно этот механизм, так как для этой реализации нет нужды в специфичной аппаратной поддержке.

2. **Аппаратное сэмплирование** - должно поддерживаться платформой. Например, новые MIPS-процессоры содержат специальный регистр PCSAMLE, позволяющий сэмплировать счётчик процессорных тактов. Преимущество этого подхода в том, что накладные расходы становятся ещё меньше, а также нивелируются недостатки реализации системных прерываний.

Впрочем, у статистически профилировщиков есть и недостатки. Ясно, что сам результат будет статистическим, а значит, в специфичных ситуациях замеры могут производиться в неподходящее время, упуская некоторую существенную информацию. Например, таким неподходящим временем может быть исполнение непрерываемого участка кода.

Как правило, конкретные реализации этого подхода имеют свои, менее очевидные недостатки, что было показано в одной из курсовых работ прошлых лет [5].

1.2 Инструментирование

Второй подход - это инструментирование. Идея заключается в изменении программного кода вставкой специальных инструкций для сбора необходимой информации. Понятно, что изменение программного кода влечёт за собой изменение поведения программы, однако, данный подход не является статистическим и нечувствителен к непрерываемости кода. Само собой, влияние на поведение программы зависит от того, какая именно информация собирается - например, сбор информации о вызовах функций вызовет куда меньше накладных расходов, чем сбор информации об исполнении каждой инструкции.

Такие инструменты хороши для программ уровня ядра.

Инструментирование может быть разделено на несколько типов:

1. **Вручную** - программисту необходимо собственноручно вставлять команды, собирающие нужную информацию.

2. **На уровне исходного кода** - специальный инструмент автоматически изменяет код программы. Пример: Parasoft Insure++

3. **На уровне промежуточного кода** - актуально при использовании языков высокого уровня. Пример: OpenPAT.

4. **С помощью компилятора** - профилировщик сам компилирует программу. Примеры: gprof, Quantity.

5. **Инструментирование бинарного файла** - производится изменение уже скомпилированного исполняемого файла.

6. **Во время исполнения** - код инструментируется прямо перед исполнением. Исполняемая программа должна находиться под полным контролем профайлера. Примеры: Pin, Valgrind, DynamoRIO.

2 Постановка задачи

Практическая часть моей курсовой работы направлена на создание профилирующего инструмента для операционной системы реального времени Embox, который позволил бы собирать информацию об исполнении программ, причём необходима не только информация о работе прикладного ПО непосредственно, но и информация о работе функций ядра во время исполнения программы.

Для такого инструмента необходима высокая точность замеров, возможность замерить время работы произвольного участка кода, в том числе и непрерываемого, а также удобный интерфейс. Точность замеров приводит к решению выбирать именно инструментирование, так как решение должно подходить в том числе и для уровня ядра ОС.

Для демонстрации работы этого инструмента должен быть проведёт ряд контрольных замеров времени работы различных системных средств, таких как создание потока, переключение между потоками, выделение памяти и так далее, а также ряд замеров работы приложений уровня пользователя, например, gzip.

3 Что было в Embox раньше

При разработке средства профилирования было принято решение сначала изучить существующий механизм профилирования в данной операционной системе реального времени и действовать далее в соответствии с текущим положением дел.

Двумя годами ранее был реализован механизм[4], позволяющий расставлять `trace_point` и `trace_block` - специальные конструкции - для измерения количества исполнений определённой команды и для замера времени работы нужных участков кода. Этот механизм действует, с помощью него можно производить замеры.

Впрочем, имеется и ряд недостатков. Один из ключевых - негибкость, то есть сложность в использовании. Например, в силу специфичности реализации, невозможно создать более одного `trace_block` в одной области видимости. Также для измерений использовался системный таймер, что означало довольно низкую точность (счёт на миллисекунды, что является довольно большим промежутком при работе в реальном времени). Главный же минус заключался в том, что все необходимые конструкции необходимо было расставлять вручную, что, конечно же, затрудняет анализ большого объёма кода.

4 Реализация профилирующего инструмента

Конечно же, новый инструментирующий профилировщик должен был по меньшей мере избавиться от недостатков предыдущих наработок.

Невозможность использовать больше одного `trace_block`‘а вытекала из не очень удачной реализации макроса, отвечавшего за объявление блоков, поэтому лучше остановиться подробнее на прочих аспектах.

4.1 Повышение точности

Малая точность замеров была устранена следующим образом. Ряд платформ поддерживает возможность измерять время в процессорных тактах. В архитектуре x86 для этого используется TSC (Time Stamp Counter), значение которого хранится в специальном регистре (MSR), для платформы SPARC - это TICK register. Возможность использовать счётчик тактов повысила точность на порядки.

Впрочем, есть несколько аспектов, которые нужно учитывать при использовании подобных счётчиков.

1. Out-of-order Execution.

   Внеочередное исполнение инструкций (Out-of-order Execution) может как увеличить, так и уменьшить показатель времени исполнения инструкций.

   Рассмотрим программу:

   ```
   rdtsc ; Считывание счётчика
   mov time,eax ; Сохранение значения
   fsqrt ; Извлечение корня
   rdtsc ; Второе считывание счётчика
   sub eax,time ; Нахождение разницы
   ```

   Например, второй замер может быть произведён до вычисления квадратного корня. Соответственно, полученное значение не будет зависеть от времени исполнения инструкции fsqrt. Для избежания этого эффекта нужно использовать упорядочивающие инструкции (serializing instructions), которые гарантируют очистку процессорного конвеера перед их выполнением. Примеры таких инструкций: cpuid для x86, SYNC для MIPS.

2. Кэширование

   Использование счётчика процессорных тактов для измерений может приводить к существенно разным результатам при повторном исполнении программы (в зависимости от того, находятся ли необходимые данные в кэше или нет), поэтому при изучении поведения программы нужно это учитывать. Полностью избавиться от такого эффекта можно только для маленьких участков кода. Для этого достаточно воспользоваться техникой “cache warming”, суть которой заключается в следующем: нужно исполнить инструкции без замера (необходимые данные загрузятся в L1-кэш), а после этого - исполнить их ещё раз, но уже замерив время. Для этого можно использовать цикл или вызов функции два раза подряд.

3. Переключение контекстов

   При профилировании больших участков кода нужно принять во внимание переключение контекстов, которое, конечно, повлияет на результат измерений. Для того, чтобы избежать этого, можно повышать приоритет изучаемого процесса.

4. Многоядерные процессоры

   Дело в том, что счётчик тактов между ядрами не синхронизируется, а это значит, что для избежания ошибок в измерении нужно избегать миграции процесса между ядрами.

4.2 Автоматическое инструментирование

4.2.1 Инструментирование отдельных модулей

При реализации автоматического инструментирования можно сделать важное наблюдение: обычно разбиение на модули подразумевает, что используемые функции не будут очень большими по объёму. Из-за этого чаще всего необходимо замерять не время исполнения произвольного участка кода, а время работы определённых функций, либо время исполнения функций из определённого модуля. Исходя из этого предположения, можно существенно уточнить задачу.

GNU Compiler Collection (GCC) предоставляет ряд возможностей для сбора информации о вызываемых функциях. Одна из них - сборка всех объектных файлов изучаемой программы с ключом -finstrument-functions. При этом перед вызовом каждой функции будет вызываться `__cyg_profile_func_enter`, а после их завершения - `__cyg_profile_func_exit`, которые можно использовать для определения времени работы функции.

При таком инструментировании кода при помощи компилятора требуется, чтобы программистом эти функции были определены. В качестве аргументов передаётся указатель на вызываемую функцию и указатель на функцию вызывающую. На основе этой информации и генерируются `trace_block` во время исполнения программы.

Одним из решений может быть добавление нужных флагов для инструментирования сразу всех модулей системы. В этом случае следует учитывать, что есть ряд функций, которые всё же нежелательны для инструментирования: это все функции, связанные с обработкой профилировочной информации; также это ряд системных функций. Для исключения этих функций из перечня профилируемых можно использовать флаг компиляции `-finstrument-functions-exclude-function-list=func1,func2...` Однако, никаким разумным образом не получится перечислить все часто вызываемые системные функции, которые изучать не нужно, а значит, возникнут бессмысленные, но ощутимые накладные расходы.

Отсюда вытекает вывод о необходимости "точечного" инструментирования. ОСРВ Embox имеет высокую конфигурируемость, что наталкивает на идею добавить поддержку автоматического инструментирования с помощью изменения специальных конфигурационных файлов. Mybuild — это система сборки для модульных приложений, которая используется в Embox.

Для этого мной была добавлена опция @InstrumentProfiling(), которая добавляет нужные флаги компиляции для указанных исходных файлов.

Простейший конфигурационный файл, описывающий программу, состоящую из одного исходного файла, может выглядеть так:

```
package embox.cmd
   @Cmd(name = "hw", help = ,man = '"""')
   module hw {
   @InstrumentProfiling("true")
   source "hw.c"
}
```

После запуска программы `hw` можно увидеть результат профилирования с помощью команды `trace_blocks -n`. Эта программа просто перебирает все `trace_block`, созданные за время работы системы и выводит информацию о них.

4.3 Сквозное профилирование

Часто информации о работе приложения самого по себе бывает недостаточно, и хочется понимать также и поведение функций ядра ОС при работе этого приложения. В этом случае было бы глупо расставлять во всех конфигурационных файлах одну и ту же опцию.

Мной была реализована программа `tbprof`. Идея заключается в том, что `tbprof` запускает наблюдаемую программу, собирая информацию лишь о тех вызовах функций, которые происходят во время исполнения исследуемой программы (включая такие вещи, как, например, системные обработчики прерываний, переключение потоков, выделение памяти и так далее).

На первый взгляд, такое больше количество накладных расходов может слишком сильно исказить результаты измерений, но на самом деле, это искажение будет не случайной природы - оно будет детерминированным, поэтому, храня информацию о количестве вызовов функций, можно будет восстановить точную оценку времени работы функций.

Тем не менее, при неудачной реализации время работы при профилировании может вырасти на порядки (например, если программа обращается преимущество к быстрым функциям, таким как, например, abs(), но обращается к ним очень часто, в то время как сохранение профилировочной информации о вызове этих функций это не такая уж и быстрая операция). В таком случае, детерминированность сохраняется, но время выполнения становится неприемлемым.

Тем не менее, удалось добиться достаточной скорости обработки профилировочной информации налету.

4.4 Возможно ли было использовать старые функции

Старая реализация средства инструментирования (расставление необходимых конструкций вручную) справлялась с поставленной задачей, но при автоматическом инструментировании всплывают некоторые проблемы.

Одна из них заключается в том, что в прежней реализации при создании `trace_block` в соответствующем участке кода создаётся статическая локальная переменная, хранящая нужную информацию, доступ к которой осуществляется через массив указателей. Конечно, возникнут дополнительные накладные расходы по использованию памяти, но вряд ли будет слишком много разумно расставленных статических переменных.

Но ситуация меняется, если мы хотим профилировать сразу всю систему. Например, если цель - исследовать время переключения и создания потоков, подавляющее большин-

ство функций системы не будет вызвано вообще, зато накладные по использованию памяти будут очень велики, ведь для всех профилируемых функций будет храниться некоторая информация в оперативной памяти, вне зависимости от того, вызывалась ли функция в принципе.

Хотелось бы уметь генерировать соответствующие `trace_block` автоматически и лишь в случае вызова соответствующей функции (ленивая инициализация). Соответственно, хранить данные лучше не в массиве, а в более подходящей для этого структуре данных, например, хэш-массиве. Эта идея и была реализована мной. Соответственно, уменьшены накладные расходы по использованию оперативной памяти.

Фактически, пришлось отказаться от старых функций и структур хранения информации и переписать их в силу того, что накладные расходы возрастали очень сильно при профилировании всей системы сразу.

5 Проведение замеров

После того, как был готов профилирующий инструмент, были проведены замеры времени работы ряда системных функций, для этого были проведены синтетические тесты: в течении нескольких часов выполнялись простые программы, вызывающие системные функции. Я постарался выбрать наиболее интересные с точки зрения ОСРВ функции для изучения: функции для работы с потоками, с критическими секциями и с прерываниями. Также интересно было посмотреть, как ведут себя системные функции при работе некоторого приложения, которое и проводит вычисления, и что-то читает с диска, и что-то записывает. В качестве такой программы был выбран архиватор `gzip`.

В таблице 1 приведена информация о потоках. Можно заметить, что време переключения (`sched_switch`) довольно велико по сравнению с другими ОСРВ (QNX, LynxOS)[7], но было выяснено, что в замерах, проведённых для этих систем, не учитывается выбор очередного потока - речь идёт лишь о переключении контекста, которое уже сопоставимо с аналогичными показателями других систем.

Функция	Кол-во запусков	Общее время	Макс. время	Среднее время
sched_switch	21050	95924603	7096	4556
thread_create	20537	1146270959	232801	55814
thread_delete	15402	325112635	71314	21108
thread_join	5134	904901208	261056	176256

Таблица 1. Потоки. Время указано в процессорных тактах.

При измерении времени работы критических секций кода не было выявлено странного поведения.

Функция	Кол-во запусков	Общее время	Макс. время	Среднее время
critical_enter	48469	103486798	3096	2135
critical_leave	48469	103158763	2516	2190

Таблица 2. Критические секции. Время указано в процессорных тактах.

Посмотрев на время выполнения мягких прерываний (таблица 3), можно обратить внимание, что разброс между средним и максимальным временем велик. Я обратился к разработчикам ОСРВ Embox для того, чтобы разобраться, почему так происходит. Выяснилось, что планирование переключения потоков происходит с помощью таймера, который использует IRQ. Для оптимизации, если система понимает, что нужно переключать поток при очередном прерывании от таймера, переключение происходит прямо в обработчике

мягкого прерывания от таймера. Поэтому и возникает такая разница между максимальным и средним временем.

Функция	Кол-во запусков	Общее время	Макс. время	Среднее время
softirq_raise	263823	6484350312	130586	24578
softirq_dispatch	263048	4651920114	85937	17684
softirq_install	443432	995795350	79425	2245
irq_dispatch	46497	4756130408	1043243	102288

Таблица 3. Прерывания. Время указано в процессорных тактах.

6 Результат

1. Исправлены некоторые недостатки предыдущей реализации профилировщика.
2. Реализован новый профилировщик для ОСРВ Embox, инструментирующий с помощью конфигурации, а не с помощью ручного изменения исходного кода.
3. Произведён ряд замеров для системных функций, отвечающих за работу потоков, критических секций и прерываний.

7 Список литературы

1. Intel, “Using the RDTSC Instruction for Performance Monitoring“, URL: http://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf

2. GNU Make Manual, URL: https://www.gnu.org/software/make/manual/

3. GCC 4.8.2 Manual, URL: http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/

4. Profiling, Wikipedia, URL: http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29

5. Крамар А.С., Трассировки ОСРВ Embox, URL: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/345/345_Kramar_report.pdf

6. Одеров Р.С., “Исследование и тестирование семплирующего метода профайлинга”, URL: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/345/345_Oderov_report.pdf

7. QEMU Emulator user documentation, URL: http://wiki.qemu.org/download/qemu-doc.html

8. Сравнительных характеристик ОСРВ, URL: http://www.ipmce.ru/about/press/popular/rdcnews05052008/