Профилирование систем реального времени

Denis Deryugin
Saint Petersburg State University
Email: deryugin.denis@gmail.com

***Аннотация*—В данной статье анализируются особенности профилирования программного обеспечения для систем реального времени. Рассматриваются специфичные для области проблемы: многозадачность, точность замеров, кросс-платформенность, необходимость замерять время работы уровней пользователя и ядра одновременно (”сквозное” профилирование), а также ряд других. Производится сравнение различных подходов к профилированию систем реального времени. Приведены аргументы в пользу инструментирования как эффективного метода для проведения “сквозного” профилирования.**

I. Введение

Профилирование — это сбор различных характеристик работы программы, таких как время выполнения определённых участков кода, объём занимаемой программой памяти, эффективность алгоритмов предсказания условных переходов.

Понимание поведения программ становится особенно важным при изучении работы систем реального времени, так как для них, прежде всего, важна детерминированность (определённость в поведении). Одной из ключевых особенностей подобных систем, делающей затруднительным использование в них средств профилирования систем общего назначения, является необходимость сбора характеристик и их анализа не отдельного приложения, а системы в целом, то есть взаимодействие всех составных частей: приложений, библиотек, ядра ОС и даже аппаратной платформы.

Известно, что по мере роста объема кода всё сложнее становится понимать поведение как отдельных её модулей, так и поведение системы в целом. Наличие соответствующих инструментов, собирающих информацию о характеристиках системы, делают разработку и оптимизацию ПО намного проще. Реализация подобных инструментов и методов их построения с учетом специфики систем реального времени позволяет упростить разработку подобных систем, а также улучшить их качество.

В данной статье рассматривают проблемы профилирования для систем реального времени. Практическая часть статьи основана на модульной конфигурируемой операционной системе реального времени с открытым исходным кодом Embox[5]. Проект Embox существует около пяти лет, за это время было написано более 150 000 строк кода (преимущественно на языке C), реализовано множество алгоритмов от классики операционных систем, описанной в литературе, до оригинальных разработок. Кроме того проект является кросс-платформенным в нем имеется поддержка ряда процессорных архитектур: ARM, x86, MicroBlaze, MIPS, PowerPC, SPARC.

II. Обзор методов профилирования

Профилировщики уже давно не являются новшеством — подобные инструменты уже существовали по крайней мере на IBM/360 и первых UNIX-системах[7].

Говоря о классификации подходов к профилированию времени исполнения программ, можно рассматривать несколько аспектов.

II-A. Динамическое и статическое профилирования

Динамическое профилирование заключается в сборе информации во время исполнения, и это – наиболее распространённый подход. Статическое же профилирование подразумевает построение некоторой математической модели по исходному коду и последующий анализ этой модели для определения поведения программы (без её запуска).[15]

II-B. Сэмплирование и инструментирование

Оба данных подхода подразумевают динамический метод анализа программы. Сэмплирование заключается в том, чтобы с некоторой периодичностью приостанавливать работу программы тем или иным образом, производя при этом анализ текущего состояния (стэк

вызовов, объём занимаемой памяти и так далее), а затем на основе собранной информации строить статистическую модель исполнения программы[2]. Идея инструментирования заключается в изменении программы вставкой специальных инструкций для сбора необходимой информации[1].

II-B1. Сэмплирование: 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 отсутствием необходимости модифицировать код программы и относительно малыми накладными расходами.

Впрочем, у сэмплирующих профилировщиков есть и недостатки. Ясно, что сам результат будет статистическим, а это приводит к потенциально неверным суждениям о поведении программы.

Вообще, возможно, что работа некоторой функции будет занимать достаточно много процессорного времени, но в силу неудачного времени сбора информации будет казаться, что функция почти не используется. Но важнее другое: есть такие участки когда, которые в принципе нельзя анализировать подобным образом.

Речь идёт об участках кода, которые исполняются с некоторыми блокировками. Например, если профилировщик использует прерывания для приостановки работы программы, сбор статистики будет невозможным для участков кода, запрещающих прерывания во время своей работы. Если для программ уровня пользователя это может быть редкостью, то для уровня ядра это может оказаться решающим.

Как правило, конкретные реализации сэмплирующих профилировщиков имеют свои, менее очевидные недостатки[14].

II-B2. Инструментирование: Инструментированием предполагает изменение программы, и, следовательно, влечёт за собой изменение её поведения, причем это может быть существенно. С другой стороны, инструментирование предоставляет точные данные измерений, а не статистическое приближение. Так же, этот подход может использоваться для измерения в критических секциях, где прерывания недоступны.

Само собой, влияние на поведение программы зависит от того, какая именно информация собирается - например, сбор информации о вызовах функций вызовет намного меньше накладных расходов, чем сбор информации об исполнении каждой инструкции.

Такой подход хорошо подходит для профилирования на уровне ядра операционной системы[3].

Инструментирование можно производить рядом способов[2]:

- *Вручную* - программисту необходимо собственноручно вставлять команды, собирающие нужную информацию.
- *На уровне исходного кода* - специальный инструмент автоматически изменяет код программы. Пример: Parasoft Insure++
- *На уровне промежуточного кода* - актуально при использовании языков высокого уровня. Пример: OpenPAT.
- *С помощью компилятора* - профилировщик компилирует программу, вставляя необходимые конструкции уже в целевом коде. Примеры: gprof, Quantity.
- *Инструментирование бинарного файла* - производится изменение уже скомпилированного исполняемого файла.
- *Во время исполнения* - в этом случае исполняемая программа должна находиться под полным контролем профилировщика. Примеры: Pin, Valgrind, DynamoRIO.

II-C. Аппаратное и программное профилирование

При профилировании могут использоваться дополнительные аппаратные компоненты, такие как таймеры высокого разрешения, счетчики тактов и так далее, в этом случае уместно говорить об аппаратном профилировании. Такие методы достаточно широко используются в различных встроенных системах и системах реального времени, поскольку они позволяют повысить точность результатов и уменьшить накладные расходы[8].

Аппаратные профилировщики особенно часто применяются в ПЛИС. Так, например, в работе “Profiling Tools for FPGA-Based Embedded Systems: Survey and Quantitative Comparison” представлен достаточно широкий спектр подходов к профилированию с использованием FPGA[8].

Рассмотрим разницу программного и аппаратного сэмплирования.

- *Программное сэмплирование* производится с помощью системных прерываний. Большинство профилировщиков использует именно этот механизм, так как для этого нет нужды в дополнительной аппаратной поддержке.
- *Аппаратное сэмплирование* должно поддерживаться аппаратной платформой. На-

пример, новые MIPS-процессоры содержат специальный регистр PCSAMLE, позволяющий сэмплировать счётчик процессорных тактов. Ещё один вариант – использование ПЛИС для создания собственного профилировщика. Преимущества подобного подхода состоят в том, что уменьшаются накладные расходы и увеличивается точность временных замеров.

Тем не менее, использование аппаратных профилировщиков возможно не для каждой аппаратной платформы. Такой вид профилирования предполагает наличие специальных аппаратных блоков, с помощью которых можно вести профилирование, или возможность расширить аппаратную платформу собственными профилирующими блоками. Также, использование полностью аппаратных профилировщиков обходится дороже, чем программных[4].

III. Особенности профилирования систем реального времени

III-A. Многозадачность

Многопоточность и многозадачность системы — основная сложность при отладке и профилировании. Этот аспект операционной системы и делает свойство реального времени нетривиальным. Действительно, при рассмотрении однозадачной и однопоточной ОС достаточно один раз замерить время исполнения программы. В то же время, единовременное выполнение множества потоков может привести к неопределённости, и организовать работу системы правильно – задача планировщика.

III-B. Кросс-платформенность

Системы реального времени зачастую предполагают использование отличных от x86 архитектур, а именно – семейство RISC[9], поскольку поведение таких архитектур более предсказуемо. В связи с этим профилировщик должен также работать на всех необходимых платформах, и для части профилировщиков это существенно. Например, Valgrind[10] привязан к архитектуре процессора и поддерживает лишь несколько платформ.

III-C. ”Сквозное” профилирование

При разработке систем реального времени важно понимать не только поведение программ уровня пользователя (это как раз та информация, которую обычно хотят получить при профилировании), но и время выполнения функций в ядре ОС, причем именно при взаимодействии с этой самой прикладной программой. То есть в подобных системах рассматривают не только результаты поведения ядра ОС и прикладной программы в отдельности, но и поведение всей системы в целом, а значит, задача усложняется.

III-D. Использование низкоуровневого программного кода

Разработка пользовательских программ обычно подразумевает использование языка высокого уровня, однако, разработка ОС подразумевает использование низкоуровневого кода, так как не все системные функции могут быть реализованы независимо от архитектуры (например, при переключении контекста, скорее всего, придётся работать непосредственно с регистрами процессора). Эти низкоуровневые вставки усложняют задачу профилирования вызовов (например, *fork()*/*vfork()*/*exit()*/переключение потоков) либо полностью реализованы на языке ассемблера, либо используют функции, написанные на языке ассемблера. Такие участки кода могут обращаться к стеку напрямую, изменяя его, что может привести, например, к невозможности “подняться” по стеку вызовов.

III-E. Точность

Для систем реального времени особенно важна точность замеров. Что накладывает особые требования к используемым аппаратным средствам

Как правило, системные таймеры предлагают точность порядка миллисекунды, а это очень большой промежуток времени (сотни тысяч и даже миллионы тактов на современных процессорах). Поэтому во многих процессорах присутствует счётчик процессорных тактов.

Возможно использование процессорных тактов в качестве таймера, но некоторые процессоры умеют менять частоту на ходу, а значит, одинаковое количество тактов, прошедшее в то или иное время может означать разное время исполнения.

Ещё одно решение – дополнительная аппаратная поддержка. Для x86, например, существует HPET[11] – специальный аппаратный таймер, работающий независимо от процессора. Для систем реального времени, не имеющей соответствующей аппаратной поддержки, могут использовать FPGA.

Следует учитывать и следующий ряд параметров, влияющих на работу системы:

- *Out-of-order Execution*.
  Внеочередное исполнение инструкций (Out-of-order Execution) может как увеличить, так и уменьшить показатель времени исполнения инструкций
  Рассмотрим программу на языке ассемблера x86:

  ```
  rdtsc ; Считывание\счётчика_
  mov time,eax ; Сохранение\значения_
  fsqrt ; Извлечение\корня_
  rdtsc ; Второесчитывание \счётчика_
  sub eax,time ; Вычисление\разницы_
  ```

  Никто не гарантирует, что инструкции будут исполняться в том же порядке, в котором следуют в листинге. 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.

- *Кэширование*
  При повторном исполнении программы (в зависимости от того, находятся ли необходимые данные в кэше или нет), показания замеров могут отличаться, поэтому при изучении поведения программы нужно это учитывать. Не отключая кэш, 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.

IV. Профилирование систем реального времени

Как следует из вышесказанного, существует достаточно много методов профилирования и выбор того или иного метода сильно зависит от конкретных задач и возможностей, поэтому говоря о системах реального времени нужно учитывать особенности, изложенные выше.

Сэмплирование имеет существенный недостаток, а именно – низкую точность. Естественно, точность будет возрастать с уменьшением интервала прерывания профилировщика, но так же будут расти и накладные расходы. К тому же, системные таймеры имеют конечную разрешающую способность, а значит, бесконечно уменьшать интервал не получится.

Добиться более высокой точности сэмплирования без критического увеличения накладных расходов можно при помощи аппаратной поддержки. Например, на FPGA можно добиться впечатляющего повышения точности. Судя по результатам этого исследования, программный профилировщик, производящий аналогичные измерения, может ошибаться в несколько раз: при профилировании функций с глубокой рекурсией сильно возрастают накладные расходы (программа может работать в несколько раз больше), а при множественном вызове быстро выполняющихся функций такие профилировщики упускают столько вызовов, что могут ошибаться до 6 раз[8].

Впрочем, аппаратную поддержку сэмплирования предоставляют не все платформы, и даже больше, в любом случае, остаётся главная проблема сэмплирования: по своей природе метод является статистическим, а значит, значительная информация может быть не учтена.

Следовательно, для профилирования систем реального времени метод инструментирования будет более подходящим. Однако, нужно понимать, что инструментировать можно принципиально разные участки программного кода.

Например, в некоторых случаях может быть достаточно измерять время работы различных потоков – тогда нужно будет добавить соответствующие вызовы в функцию, отвечающую за переключение потоков. Так как модификация кода потребуется только в одном месте, то подойдёт и инструментирование вручную, которое проще всего реализовать, и которое не требует никакой специальной поддержки. Впрочем, есть и явный недостаток: с одной стороны, накладные расходы в таком случае будут довольно низкими, но в то же время информация, скорее всего, окажется недостаточно детальной. Действительно, будет видно, какой поток работает дольше остальных, но будет непонятно, где именно тратится время. Чем больше программного кода использует один поток, тем сложнее будет судить о том, где необходима оптимизация.

Другой крайностью может быть построчное инструментирование программного кода, но накладные расходы в этом случае будут слишком велики, поэтому от этой идеи лучше сразу отказаться. К тому же, такая детальная ин-

формация вряд ли будет иметь принципиально большую полезность, чем, к примеру, время исполнения различных функций[1].

V. Реализации инструментирующего профилировщика в ОСРВ Embox

Практическая часть данной работы была направлена на создание профилирующего инструмента для операционной системы реального времени Embox[5], который позволил бы собирать информацию о времени исполнении программ.

V-A. Требования к профилировщику

Требования были сформулированы следующим образом: профилировщик должен учитывать особенности систем реального времени, а именно:

- Необходима не только информация о работе прикладного ПО непосредственно, но и информация о работе функций ядра во время исполнения программы (”сквозное” профилирование).
- Высокая точность измерений.
- Возможность измерить время работы произвольного участка кода, вне зависимости от различных блокировок.
- Кросс-платформенность.

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.

V-B. Использование ключей компилятора

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 функция *__cyg_profile_func_enter*, а после их завершения - *__cyg_profile_func_exit* – пролог и эпилог, – которые можно использовать для определения времени работы функции. Следует отметить, что в этом случае использование ключевого слова *inline* перед описанием функций не произведёт никакого эффекта.[13]

От программиста требуется определить эти функции. В качестве аргументов им передаётся указатель на вызываемую функцию и указатель на функцию вызывающую. На основе этой информации генерируются *trace_block* во время исполнения программы.

trace_block - это специальная конструкция для замера времени работы нужных участков кода. Она имеет следующий вид:

```
struct __trace_block {
    const char *name;
    struct location_func location;

    void *func;

    struct tb_time *time_list_head;

    time64_t time;
    time64_t max_time;

    int count;
    int depth;
    bool is_entered;
    bool active;
};
```

Поля *name* и *location* необходимы при инструментировании вручную — первое позволяет использовать произвольное имя для конструкции, а второе содержит в себе информацию о том, где происходит объявление *trace_block*, соответствующие файл с исходным кодом и номер строки. Для всех *trace_block* информация собирается во время компиляции и затем уже не изменяется.

Для функций, которые необходимо профилировать с помощью упомянутых возможностей GCC, создаётся пул структур *trace_block*, который представляет из себя список, инициализируемый во время компиляции. Таким образом, при необходимости выделить структуру для очередной функции не нужно обращаться к системному механизму выделения памяти, а значит, можно исследовать поведения ядра на ранних стадиях запуска системы, когда модуль выделения памяти ещё не инициализирован.

При вызове фунций *__cyg_profile_func_enter* и *__cyg_profile_func_exit* производится поиск соответствующей записи в хэш-таблице, и если данная функция обрабатывается впервые, выделяется структура *trace_block* из упомянутого выше пула. Далее вызывается обработчик, который записывает информацию о входе в функцию, учитывая глубину рекурсии. Для этого используется список *time_list_head*, и при последующем выходе из функции для подсчёта времени, которое работала функция, используется именно это значение.

В дальнейшем достаточно перебрать все элементы из пула для того, чтобы узнать, сколько времени выполнялись различные функции. Те из них, которые не вызывались ни разу, там

присутствовать, конечно, не будут.

Таким образом, возможно собирать информацию о времени выполнения функций без ручного инструментирования. Впрочем, после этого необходимо решить, как использовать данный механизм наиболее эффективно.

Один из способов инструментирования заключается в использовании ключа *-finstrument-functions* при компиляции всех модулей системы. В этом случае следует учитывать, что есть ряд функций, которые всё же нежелательны для инструментирования: это все функци , связанные с обработкой профилировочной информации; также это ряд системных функций. Для исключения этих функций из перечня профилируемых можно использовать ключ 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 следует вывод о необходимости “точечного” инструментирования.

V-C. Mybuild

ОСРВ Embox[5] состоит из модулей и имеет высокую конфигурируемость. Описание модулей в системе сборки Mybuild отделены от исходного кода. Это позволило добавить поддержку автоматического инструментирования с помощью изменения специальных конфигурационных файлов.

Указание аннотации @InstrumentProfiling(), добавляет нужные ключи компиляции для указанных исходных файлов. Простейший конфигурационный файл, описывающий программу, состоящую из одного исходного файла, выглядит 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.

С помощью этой же аннотации можно задать список функций для профилирования в конкретной единице компиляции. Например, следующее описание включает профилирование только функции *print_usage*

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

V-D. Точность замеров

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[12]. Эти значения и используются в качестве таймера.

V-E. ”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.

Была реализована утилита *tbprof*[6]. 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 события, 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. Случиться это может, если, например, программа обращается преимущественно 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 быстрая операция. В таком случае, детерминированность сохраняется, но время выполнения становится неприемлемым. Эту проблему удалось частично решить, поскольку при реализации профилировщика применялась

хэш-таблица, и длительная операция выделения данных под профилирующую информацию производилась только при первом обращении. Таким образом удалось добиться достаточной скорости обработки профилировочной информации ”налету”.

VI. Проведение замеров

Были проведены замеры времени работы ряда системных функций под различной нагрузкой. В течение 180 минут выполнялись: архиватор из проекта с открытым кодом zlib, набор модульных тестов из состава Embox. Архиватор является типичным приложением, работающим с файлами. Набор модульных тестов воспроизводит различные ситуации для вызова наиболее интересных с точки зрения ОСРВ функций: для работы с потоками, с критическими секциями и с прерываниями.

В таблице 1 приведена информация о потоках. Можно заметить, что время переключения (*sched_switch*) достаточно велико велико, 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	Запуски	Макс. время	Ср. время
sched_switch	21050	7096	4556
thread_create	20537	232801	55814
thread_delete	15402	71314	21108
thread_join	5134	261056	176256

Таблица 1. Потоки. Время указано в процессорных тактах.

В таблице 2 приведены данные о времени выполнения критических секций кода. Как видно из этих данных разброс значений минимальный. Это вызвано прежде всего тем, что отсутствует влияние от переключения потоков.

Функция	Запуски	Макс. время	Ср. время
critical_enter	48469	3096	2135
critical_leave	48469	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. Дальнейшее иследование вопроса показало, что во время обработки мягкого прерывания от системного таймера может происходить вызов длительной процедуры перепланирования потоков. Поэтому для корректной оценки времени работы функций внутри которых могут опционально вызываться функции, работающие гораздо дольше, необходим механизм оценки влияния вложенных функций на исследуемую. На данном этапе такой механизм не был реализован, но планируется в дальнешей работе. 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	Запуски	Макс. время	Ср. время
softirq_raise	263823	130586	24578
softirq_dispatch	263048	85937	17684
softirq_install	443432	79425	2245
irq_dispatch	46497	1043243	102288

Таблица 3. Прерывания. Время указано в процессорных тактах.

VII. Заключение

Профилирование – сложная задача сама по себе. При профилировании систем реального времени необходимо учитывать ряд особенностей, которые делают затруднительным или вообще не возможным использование привычных для систем общего назначения инструментов и подходов.

Общая проблема для всех профилировщиков — чем выше точность, тем больше накладных расходов. Из-за ряда ограничений для систем реального времени наиболее подходит инструментирование функций для сбора информации о времени исполнения программного кода.

Приведён пример архитектурно-независимой реализации инструментирующего профилировщика на основе ОСРВ ”Embox” и представлены примеры его работы.

Список литературы

[1] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. In Proc. of ACM SIGPLAN Symposium on Compiler Construction, pages 120–126, Boston, Mass., 1982. ACM.

[2] Thiel J. An overview of software performance analysis tools and techniques: From gprof to dtrace //Washington University in St. Louis, Tech. Rep. – 2006.

[3] Nishioka S., Kawaguchi A., Motoda H. Process-labeled kernel profiling: a new facility to profile system activities //Proceedings of the 1996 annual conference on USENIX Annual Technical Conference. – Usenix Association, 1996. – С. 24-24.

[4] Awadalla M. H. A., El-Deen K. E. Real-Time Software Profiler for Embedded Systems. http://ijsce.org/attachments/File/v3i1/A1286033113.pdf

[5] Embox, операционная система реального времени https://code.google.com/p/embox/

[6] Исходный код утилиты tbprof https://code.google.com/p/embox/source/browse/trunk/embox/src/cmds/profiler/tbprof.c

[7] Wikipedia, ”Computer profiling”, http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29, дата обращения: 21 августа 2014.

[8] Jason G. Tong, Mohammed A. S. Khalid, “Profiling Tools for FPGA-Based Embedded Systems: Survey and Quantitative Comparison”, Journal of Computers, Vol 3, No 6 (2008), 1-14, Jun 2008.
[9] Wikipedia, ”Reduced Instruction Set Computer”, http://en.wikipedia.org/wiki/Reduced_instruction_set_computing, дата обращения: 21 августа 2014.
[10] Valgrind, список поддерживаемых платформ, http://valgrind.org/info/platforms.html, дата обращения: 10 августа 2014.
[11] Intel, IA-PC HPET (High Precision Event Timers) Specification, 2004.
[12] Oracle, Oracle SPARC Architecture 2011, 2011.
[13] GCC, руководство программиста, https://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/Code-Gen-Options.html, дата обращения: 27 августа 2014.
[14] Одеров Р.С., Исследование и тестирование семплирующего метода профайлинга на примере профилировщика производительности Intel VTune Amplifier XE, http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/345/345_Oderov_report.pdf, 2012.
[15] Edison Mera, Pedro Lopez-Garciaa, German Puebla, Manuel Carro and Manuel Hermenegildo, Combining Static Analysis and Profiling for Estimating Execution Times in Logic Programs, http://clip.dia.fi.upm.es/papers/estim-exec-time-tr.pdf, 2006