Санкт-Петербургский государственный университет

Программная инженерия

Остроухов Антон

Оптимизация процесса тестирования OpenJDK для встраиваемых платформ

Бакалаврская работа

Научный руководитель:

д.ф.-м.н., профессор А. Н. Терехов

Консультант:

ассистент каф. системного программирования А. П. Козлов

Рецензент:

ведущий инженер-программист ООО «Азул Системс» А. В. Красный

Санкт-Петербург

2020

SAINT PETERSBURG STATE UNIVERSITY
Software Engineering

Anton Ostrouhhov

Optimization of an OpenJDK testing process for embedded platforms

Bachelor's thesis

Scientific supervisor:
professor Andrey Terekhov

Scientific consultant:
assistant Anton Kozlov

Reviewer:
senior software engineer at Azul Systems Andrey Krasnyy

Saint Petersburg
2020

Введение **4**

1. Постановка задачи **7**

2. Обзор **8**

2.1 Программная модель FPU на ARMv7 8

2.2 Поддержка FPU в ядре Linux 9

3. Возможные решения **10**

3.1 Образ ядра Linux без модуля поддержки FPU 10

3.2 Изменение состояния FPU между запусками системы 10

3.3 Изменение состояния FPU во время выполнения 10

3.4 Выводы 11

4. Реализация переключения состояния FPU во время выполнения **14**

4.1 Базовая архитектура 14

4.2 Поддержка встроенной оптимизации переключения контекста 15

4.3 Утилита переключения состояния FPU 17

5. Апробирование **19**

Заключение **22**

Список литературы **23**

Введение

Java - один из самых популярных языков программирования в мире. Это накладывает на сообщества и компании, занимающиеся реализацией и сборкой дистрибутивов платформы Java, требование обеспечения высокого качества поставляемых продуктов.

Проект OpenJDK - это открытая эталонная реализация стандартной версии платформы Java (англ. Java Standard Edition, сокр. Java SE), спецификации языка программирования Java. Многие популярные реализации платформы Java основаны на проекте OpenJDK.

В проекте OpenJDK существует поддержка платформ с процессорной архитектурой ARMv7 и ядром Linux. ARMv7 является распространенной архитектурой, на основе которой делаются процессоры, широко используемые в смартфонах, роботах, маршрутизаторах, одноплатных компьютерах Raspberry Pi и прочих встраиваемых системах. На самом деле эта архитектура порождает три разные платформы, различие которых заключается в способе работы с числами с плавающей запятой и математическим сопроцессором (англ. Floating Point Unit, сокр. FPU).

1. В soft системах FPU нет, вычисления чисел с плавающей запятой проходят на центральном процессоре с помощью библиотечных функций.
2. В softfp системах FPU может присутствовать, но используется только библиотечными функциями для оптимизации их работы, что позволяет запускать на таких системах и исполняемые модули, которые были скомпилированы для soft систем.

3. В hard системах FPU обязателен, а исполняемые модули сами вызывают FPU-специфичные инструкции для работы с числами с плавающей запятой.

В целях уменьшения количества дистрибутивов OpenJDK, одна и та же сборка может эффективно использовать soft и softfp системы. Это достигается за счёт интенсивного использования кодогенерации во время запуска виртуальной машины Java, в ходе которого может быть порождена максимально эффективная среда для исполнения на конкретной аппаратной реализации. С другой стороны, это усложняет вариативность тестируемых конфигураций. Несмотря на один дистрибутив, реализация должна быть протестирована на soft и softfp системах.

Реализация Java для какой-либо платформы накладывает высокие требования совместимости, которые достигаются благодаря обширному тестированию сборок на каждой платформе. Например, в OpenJDK 11[1] тестовая структура[2] содержит более 30000 тестов [1]. Полный цикл тестирования проходит приблизительно за два дня на одноплатных компьютерах Raspberry Pi 2 Model B[3]. Необходимость поддерживать как soft, так и softfp системы приводит к повышенной стоимости выпуска очередных версий дистрибутива на базе OpenJDK. Также жёсткое разделение на soft и softfp системы не позволяет динамически распределять нагрузку тестирования, что особенно важно при выпуске одиночных специализированных сборок.

В данной работе исследуется возможность использования softfp систем с

[1] Открытая эталонная реализация Java SE версии 11.
[2] Директория «test» репозитория «jdk11» проекта OpenJDK.
[3] Имеют процессор ARM Cortex-A7 (см. [2]).

наличием FPU как soft систем, где его использование должно быть запрещено. Такая возможность должна унифицировать множество поддерживаемых конфигураций, тем самым упростив их поддержку, а также позволит ускорить процесс тестирования OpenJDK для soft конфигурации, так как его можно будет проводить с использованием softfp вычислительных ресурсов.

1. Постановка задачи

Целью данной работы является реализация возможности отключения и включения математического сопроцессора в ядре Linux для процессорной архитектуры ARMv7. Для достижения этой цели в рамках работы были сформулированы следующие задачи:

1) сделать обзор программной модели FPU на процессоре ARM Cortex-A7, реализующем архитектуру ARMv7;
2) рассмотреть возможные способы смены режима работы FPU;
3) реализовать выбранный способ;
4) опробовать решение в существующем процессе тестирования.

2. Обзор

2.1 Программная модель FPU на ARMv7

Архитектура ARMv7 имеет опциональное расширение для поддержки плавающей точки (англ. Floating-point Extension) [3]. Другими словами, FPU может как присутствовать, так и отсутствовать на системах, реализующих архитектуру ARMv7. При попытке вызова инструкции, специфичной для FPU, на системе с отсутствующим FPU, процессор сгенерирует исключение неопределенной инструкции (англ. Undefined Instruction)

Среди служебных регистров FPU есть регистр исключения плавающей точки (англ. Floating Point Exception Register, сокр. FPEXC), целью которого является предоставление контроля над FPU [4]. На рис. 1 изображено строение этого регистра.

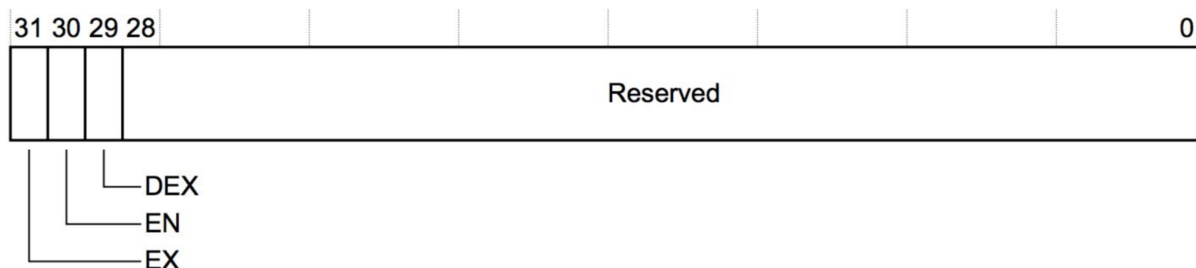

Рис. 1: Устройство регистра FPEXC, представленное в [4].

Бит EN отвечает за включение и выключение FPU. При запуске бит EN равен «0», что означает выключенное состояние FPU. При таком значении бита EN попытки вызвать большинство инструкций (за исключением некоторых служебных) будут вызывать исключение неопределенной инструкции. Точно

такое поведение можно наблюдать на системах, где FPU отсутствует физически.

Операционная система должна реализовать первичную инициализацию и сопровождение работы FPU.

2.2 Поддержка FPU в ядре Linux

Ядро Linux имеет модуль, отвечающий за поддержку FPU в процессорах архитектуры ARMv7. Если модуль включен, то ядро будет поддерживать FPU и позволит с ним работать.

Этот модуль включается присвоением значения «Y» параметру CONFIG_VFP в конфигурационном файле ядра перед его сборкой. Также CONFIG_VFP используется для условной компиляции блоков в других частях ядра, которым необходимо взаимодействовать с модулем ядра FPU. Существование такого параметра облегчает поиск по коду ядра с целью модификации логики использования FPU. Конфигурационный файл затем будет использован файлом сборки (англ. Makefile) в момент запуска процесса сборки ядра Linux [5].

Модуль представляет из себя набор исходных файлов, написанный на языке программирования C.

3. Возможные решения

В качестве возможных путей решения поставленной задачи можно выделить следующие подходы.

3.1 Образ ядра Linux без модуля поддержки FPU

Возможным решением поставленной задачи может служить замена ядра Linux в softfp системах на аналогичное, но с выключенным модулем поддержки FPU. Это означает, что при запуске операционной системы не будет производиться инициализация FPU, а именно не будет производиться запись значения «1» в бит EN регистра FPEXC. Таким образом устройство станет полноценной soft системой.

3.2 Изменение состояния FPU между запусками системы

Другим решением может быть переключение состояния FPU между перезапусками системы посредством записи желаемого состояния в заранее определенное место для конфигурации. Перед запуском ядро Linux будет считывать желаемое состояние FPU. Если желаемым состоянием будет наличие FPU, то ядро Linux запустит модуль, отвечающий за инициализацию и сопровождение FPU. В противном случае модуль не будет запущен.

3.3 Изменение состояния FPU во время выполнения

В качестве модификации предыдущего решения можно рассмотреть переключение состояния FPU во время выполнения ядра Linux. Во время

переключения контекста[4] (англ. context switch) будет происходить чтение желаемого состояния FPU для процесса, на который происходит переключение. Далее, в зависимости от желаемого процессом состояния FPU, ядро будет выставлять соответствующее значение бита EN регистра FPEXC. Несмотря на инициализацию и возможную нормальную работу FPU во время исполнения предыдущего процесса, при нулевом значении бита EN регистра FPEXC вызов FPU-специфичной инструкции вызовет исключение неопределенной инструкции.

3.4 Выводы

Первое решение (образ ядра Linux без модуля поддержки FPU) возможно реализовать достаточно быстро. Нужно собрать ядро Linux с конфигурацией, в которой параметр CONFIG_VFP равен «N». Исходный код ядра не потребует изменений. Однако, для переключения между soft и softfp состояниями придется иметь два образа ядра Linux: одно с поддержкой FPU, второе без его поддержки. Нужно будет заменить одно ядро другим для смены состояния. Тем не менее, такой подход позволяет использовать одно устройство и как soft, и как softfp систему.

Второе решение (изменение состояния FPU между запусками системы) требует небольшого вмешательства в исходный код ядра Linux. Плюсом перед предыдущим решением является отсутствие необходимости иметь два образа ядра и тратить время на замену одного другим.

Первое и второе решения имеют существенный недостаток - необходимость

[4] Переключение с одного процесса на другой, требующее обновления аппаратуры (регистров процессора и прочего) до состояния, соответствующего новому процессу (см. [6]).

перезагружать систему для изменения ее состояния. Некоторые встраиваемые системы спроектированы таким образом, что их можно перезагружать только вручную. Более того, после каждого перезапуска системы существует вероятность, что тестовая машина не сможет должным образом зарегистрироваться в общей тестовой инфраструктуре. Причиной могут являться, например:

- измененный и не сохраненный перед перезагрузкой системы блок файловой системы (например, см. [7][8]);
- сбои корневой файловой системы;
- остаточное состояние в тестовой инфраструктуре.

Каждый такой сбой потребует ручного вмешательства инженера. Изменение конфигурации системы увеличивает количество записей на флеш-память, что приводит к ускорению её износа. Её замена также является действием, выполняемым вручную. Оба способа не являются целесообразными из-за подверженности ошибкам и существенному количеству действий, выполняемых вручную.

Третий подход (изменение состояния FPU во время выполнения) снимает ограничение на необходимость перезапуска системы при желании сменить состояние FPU. Переключение состояния FPU будет происходить мгновенно. Более того, можно будет запускать разные soft и softfp процессы одновременно, обеспечивая гибкое распределение ресурсов. Например, станет возможно выделить часть ядер центрального процессора для запуска soft процессов, а остальные предоставить для softfp процессов.

Единственный минус третьего решения - сравнительно сложная реализация

и большая модификация ядра Linux (относительно двух других решений). Однако в последствии может быть выиграно время не только на тестировании сборки OpenJDK, но и время обслуживания тестировочной лаборатории. Тестирование ускорится, но количество soft систем не увеличится.

Возможность изменения состояния FPU во время выполнения позволит быстро динамически распределять вычислительные ресурсы между softfp и soft задачами без побочных эффектов. Именно это решение следует реализовывать для выполнения поставленной цели.

4. Реализация переключения состояния FPU во время выполнения

Для поддержки возможности изменения состояния FPU во время выполнения следует изменить и дополнить исходный код модуля поддержки FPU для процессорных архитектур ARM в ядре Linux.

4.1 Базовая архитектура

Возможность каждого процесса определять желаемое для себя состояние FPU требует изменения общей структуры процесса в ядре Linux. Она должна содержать поле, отображающее разрешение использования FPU для процесса.

Файловая система procfs позволяет передавать информацию из пространства пользователя в пространство ядра и наоборот путем предоставления интерфейса к структурам данных ядра [9]. На этапе инициализации ядра создается специальное вхождение в файловой системе procfs, которое называется «vfpStatus». Пользователь задает запрет на использование FPU для текущего процесса путем записи значения «0» в «vfpStatus». Запись значения «1» даёт разрешение на использование FPU текущим процессом. Затем запись обрабатывается в пространстве ядра: в поле разрешения использования FPU структуры текущего процесса записывается, соответственно, значение «0» или «1». FPU выключается (включается) путем записи значения «0» («1») в бит EN регистра FPEXC. Далее значение бита EN выставляется равным значению поля разрешения FPU структуры процесса каждый раз при переключении контекста.

При чтении пользователем специального вхождения «vfpStatus» файловой системы procfs в пространстве ядра происходит чтение поля разрешения использования FPU из структуры текущего процесса и его передача в пространство пользователя. Это значение отображает разрешение или запрет использования FPU текущим процессом.

Первому процессу в операционной системе (инициализация, англ. initialization, сокр. init) разрешается использовать FPU. Каждый новый процесс наследует свойство разрешения на использование FPU у родительского. Затем каждый процесс может изменить это свойство только для себя.

Вся описанная архитектура была реализована в виде модификаций модуля поддержки FPU для процессорных архитектур семейства ARM в ядре Linux. При проверке решения, тем не менее, оказалось, что оно не всегда работает так, как ожидается.

4.2 Поддержка встроенной оптимизации переключения контекста

В ядре Linux реализована оптимизация переключения контекста за счет переключения контекста FPU только непосредственно перед его использованием. Наличие программного кода этой оптимизации и его взаимодействие с модулем поддержки FPU не задокументировано. Такая особенность переключения контекста изменяла ожидаемое поведение изначального функционала реализованного расширения.

Возможность использования данного подхода описана в [10]. Подход заключается в выключении FPU при переключении контекста и его включении только по необходимости. Во время переключения контекста FPU выключается

(значение бита EN становится равным «0»). При следующем вызове FPU-специфичной инструкции происходит исключение неопределенной инструкции, которое обрабатывается модулем поддержки FPU - FPU включается (значение бита EN становится равным «1»), а вызвавшая исключение инструкция повторяется для нормального исполнения. Более того, FPU всегда включается при порождении нового процесса. Такой подход экономит время на переключении контекста FPU. Выключение FPU позволяет не совершать переключение его контекста для каждого процесса. Более того, выключенный FPU сохраняет контекст последнего использовавшего его процесса, тем самым позволяя не менять состояние регистров при следующем его использовании этим процессом (учитывая, что во время простоя этого процесса FPU не использовался никаким другим).

Для полноценной работы базовая архитектура разработанного решения была дополнена поддержкой встроенной оптимизации переключения контекста в ядре Linux. При переключении контекста FPU всегда выключается, работа оптимизации не нарушается. При следующем вызове FPU-специфичной инструкции и последующей обработке исключения неопределенной инструкции сначала происходит проверка того, что у процесса, из которого была вызвана инструкция, в поле разрешения использования FPU стоит значение «1». Если это так, то FPU включается и вызвавшая исключение инструкция повторяется для нормального исполнения (обычная работа оптимизации). В противном случае FPU не включается, инструкция не

повторяется и исключение неопределенной инструкции преобразуется в сигнал SIGILL[5] для текущего процесса.

4.3 Утилита переключения состояния FPU

Для удобства использования реализованного решения была создана утилита chvfp. При ее запуске происходит запись указанного пользователем состояния FPU в специальное вхождения «vfpStatus» файловой системы procfs («1» для разрешения использования FPU, «0» для запрета) и запуск указанной программы. В листинге 1 представлен пример вызова пользователем приложения, которое запускает тестирование сборки OpenJDK в soft конфигурации.

```
$ chvfp 0 /home/pi/testharness
Running JTReg test...
runner starting test: java/lang/ToString.java
runner finished test: java/lang/ToString.java
Passed. Execution successful
```

Листинг 1: Начало тестирования сборки OpenJDK в конфигурации soft

Таким образом, для тестирования сборки OpenJDK в soft конфигурации достаточно запустить программу testharness с использованием утилиты chvfp со значением «0». Все порождаемые программой testharness процессы будут наследовать от исполняющего ее процесса запрет на использование FPU. Параллельно (либо после окончания тестирования soft конфигурации) можно

[5] Запрещенная инструкция (англ. illegal instruction).

запустить тестирование сборки OpenJDK в softfp конфигурации (с возможностью использовать FPU), запустив другой экземпляр программы testharness без использования утилиты chvfp (разрешая использование FPU).

5. Апробирование

Реализованное решение было опробовано в существующем процессе тестирования сборок OpenJDK в лаборатории компании Азул Системс.

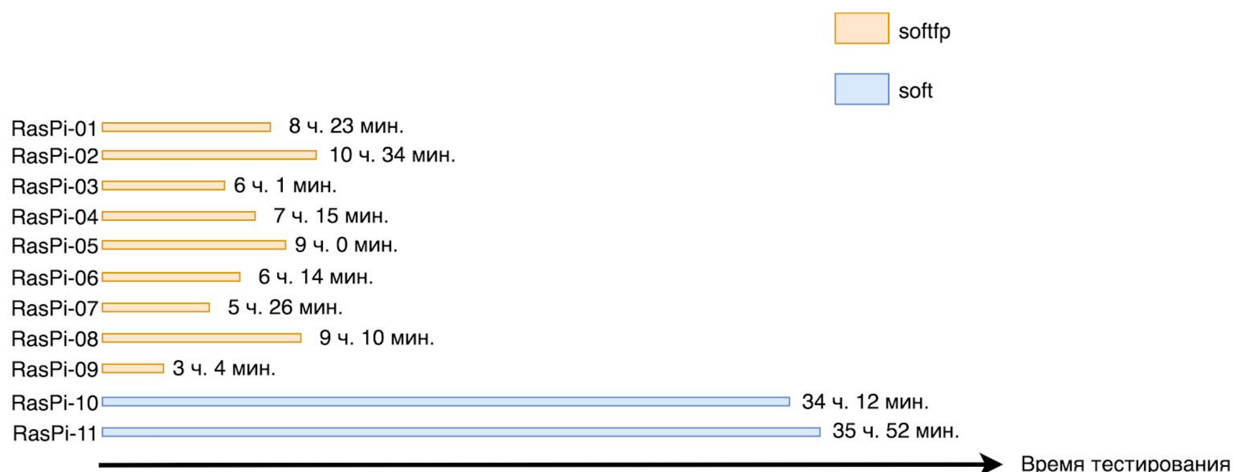

Рис. 2: График процесса тестирования сборки OpenJDK до изменений

На рис. 2 представлен график, отображающий изначальный процесс тестирования сборки OpenJDK на soft и softfp системах. Для тестирования имеется 9 softfp машин и 2 soft машины. На графике видно время занятости каждой машины, участвующей в тестировании. Тестирование представляет из себя множество задач, каждая задача тестирует определенный компонент сборки OpenJDK. Машина берет задачу, выполняет ее, затем берет следующую. Разные задачи занимают разное время, поэтому общее время занятости каждой машины разное. Тестирование на девяти softfp системах полностью завершилось через 10 часов 34 минуты, а на двух soft системах все тестирование завершилось через 35 часов 52 минуты. После того, как все

задачи на тестирование сборки OpenJDK в softfp конфигурации закончились, девять softfp машин простаивали без работы.

Рис. 3: График процесса тестирования сборки OpenJDK после изменений

На рис. 3 представлен график, отображающий процесс тестирования с использованием ядра Linux, содержащего реализованное решение. Задачи на тестирование в softfp режиме получают приоритет. Когда все такие задачи завершаются, запускаются задачи для тестирования soft конфигурации. Каждая задача получает необходимое состояние FPU от родительского процесса, которым является процесс с программой, подготавливающей тестовое пространство и обеспечивающей среду выполнения тестов. Эта программа считывает тип задачи (soft или softfp) из конфигурационных файлов тестирования перед ее запуском, а затем запускает утилиту chvfp с соответствующим параметром.

С использованием модифицированного ядра Linux тестирование сборки OpenJDK в soft конфигурации было полностью завершено через 14 часов 56 минут. Общее время тестирования в обеих конфигурациях (и время

тестирования в soft конфигурации) сократилось на 20 часов 56 минут (58%). Стоит отметить, что время полного тестирования сборки OpenJDK на softfp системах не увеличилось (около 10 часов 30 минут), что говорит о том, что привнесенные в ядро изменения не ухудшили существующий процесс тестирования.

Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Сделан обзор программной модели FPU на процессоре ARM Cortex-A7, реализующем архитектуру ARMv7. Найден способ управления состоянием FPU.
2. Рассмотрены три возможных способа смены режима работы FPU, указаны их достоинства и недостатки. Выведено оптимальное решение.
3. Реализован способ смены режима работы FPU во время выполнения в ядре Linux, решение учитывает все особенности и оптимизации.
4. Решение опробовано в существующем процессе тестирования. Опробованное решение ускорило общий процесс тестирования на 58%.

Набор разработанных в рамках данной работы изменений ядра Linux можно найти по ссылке: https://github.com/aostrouhhov/vfp-switch.git

Список литературы

1. OpenJDK Mercurial Repositories. [Электронный ресурс]. URL: https://hg.openjdk.java.net/jdk/jdk11
2. BCM2836 - Raspberry Pi Documentation. [Электронный ресурс]. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md
3. ARM Architecture Reference Manual. [Электронный ресурс]. URL: https://static.docs.arm.com/ddi0406/cd/DDI0406C_d_armv7ar_arm.pdf
4. Cortex-A7 Floating-Point Unit. Technical Reference Manual. [Электронный ресурс]. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0463f/DDI0463F_cortex_a7_fpu_r0p5_trm.pdf
5. Linux Kernel Makefiles - The Linux Kernel documentation. [Электронный ресурс]. URL: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html
6. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. - СПб.: Питер, 2015 - 1120 с.
7. FAT-fs dirty bit problem - Raspberry Pi Forums. [Электронный ресурс]. URL: https://www.raspberrypi.org/forums/viewtopic.php?t=73284

8. reboot system bring dirty bit to boot partition - Raspberry Pi Forums. [Электронный ресурс]. URL: https://www.raspberrypi.org/forums/viewtopic.php?t=139135
9. Linux Programmer's Manual. [Электронный ресурс]. URL: http://man7.org/linux/man-pages/man5/proc.5.html
10. Application Note 98. VFP Support Code. [Электронный ресурс]. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0098a/index.html