**Отчет Yadro. Этап 1**

**Изучение технической литературы по сборке и профилированию проекта Yocto, разработка архитектуры инструмента профилирования и анализа**

# Введение

Yocto Project является открытым проектом, предоставляющим набор инструментов и методик для создания образов Linux для встраиваемых систем. Проект Yocto позволяет создавать настраиваемые дистрибутивы Linux, оптимизированные для конкретного оборудования и требований приложений.

Профилирование является важной задачей при разработке встраиваемых систем с использованием Yocto, так как позволяет выявить узкие места и в последствии оптимизировать производительность и использование ресурсов в процессе сборки. В рамках данного исследования изучались существующие инструменты профилирования, предоставляемые проектом Yocto, а также разрабатывалась архитектура собственного инструмента профилирования и анализа.

# Постановка задачи

В настоящее время сборка образа при помощи Yocto занимает около 2-х часов. Это недопустимо долго при необходимости часто собирать разные образы. Для того, чтобы оптимизировать и ускорить сборку, необходимо разобраться в причинах, по которым она проходит так долго.

Целью работы на данном этапе является изучение технической литературы по сборке и профилированию проекта Yocto и разработка архитектуры инструмента для профилирования и анализа.

# Сборка образа с использованием Yocto Project

# Концепция сборки

Сборка проходит при использовании распараллеливания. Время сборки сильно зависит от аппаратных характеристик компьютера, на котором она происходит.

Bitbake – это инструмент для сборки, используемый в Yocto Project. Bitbake использует описания рецептов для сборки и установки пакетов, а также для создания образов дистрибутива.

Рассмотрим основные составляющие концепции сборки:

1. Рецепты (задания):

Задания BitBake, хранящиеся в файлах .bb, служат базовыми метаданными, обеспечивая BitBake:

* описания пакетов (автор, домашняя страница, лицензия и т. п.);
* версии заданий;
* имеющиеся зависимости (при сборке и работе);
* местоположение исходного кода и способ его извлечения;
* потребность в применении правок, их местоположение и способ применения;
* детали настройки и компиляции исходного кода;
* место установки программ на целевой машине. В контексте BitBake или использующей программу системы сборки файлы .bb считаются заданиями. Иногда задания называют пакетами, однако это не совсем корректно, поскольку одно задание может включать множество пакетов.

1. Конфигураторы (файлы конфигурации)

Конфигурационные файлы .conf задают переменные, управляющие процессом сборки. Эти файлы делятся на несколько категорий, задающих конфигурацию машины и дистрибутива, опции компиляции, базовые и пользовательские параметры. Основным файлом конфигурации служит bitbake.conf в каталоге conf дерева источников.

1. Классы

Файлы классов .bbclass содержат информацию, которая может совместно использоваться множеством файлов метаданных. Классы из base.bbclass включаются автоматически для всех заданий и классов и содержат определения стандартных базовых задач, таких как выборка, распаковка, настройка (по умолчанию пуст), компиляция (запускается при наличии Makefile), инсталляция (по умолчанию пуст) и подготовка пакетов (по умолчанию пуст). Эти задачи часто переопределяются или расширяются другими классами, созданными при разработке проектов.

1. Слои (уровни)

Уровни позволяют разделить разные типы настроек. Может показаться заманчивым держать все в одном месте при работе над проектом, однако модульная организация существенно упростит внесение изменений в проект. Для иллюстрации применения уровней рассмотрим конфигурацию для конкретной целевой машины. Этот тип настройки обычно выделяют в специальный уровень, называемый уровнем BSP1. Настройки для машины следует изолировать от заданий и метаданных, поддерживающих, например, новую среду GUI. Однако важно понимать, что уровень BSP может включать машинозависимые дополнения для заданий GUI без вмешательства в этот уровень. Это делается с помощью файлов дополнения BitBake append (.bbappend).

1. Файлы дополнения

Файлы дополнения .bbappend расширяют и переопределяют информацию имеющегося файла задания. BitBake считает, что каждому файлу дополнения соответствует своё задание, имена дополнения и задания должны иметь общий корень и могут различаться лишь суффиксом (например, formfactor\_0.0.bb и formfactor\_0.0.bbappend). Информация в файле дополнения расширяет или переопределяет содержимое соответствующего файла задания. В именах файлов дополнения можно применять шаблон %, позволяющий дополнять множество заданий сразу. Например, файл busybox\_1.21.%.bbappend будет соответствовать заданиям busybox\_1.21.x.bb разных версий.

## Общий ход работы BitBake

1. Парсинг .bb файлов: BitBake парсит и анализирует .bb файлы для определения задач и соответствующих функций, которые должны быть выполнены.
2. Строительство дерева зависимостей: на основе информации из .bb файлов, BitBake строит дерево зависимостей, отображающее связи между задачами и рецептами.
3. Построение очереди выполнения: затем BitBake формирует очередь выполнения задач. Очередь состоит из объектов класса RunTaskEntry. В каждом таком объекте хранится следующая информация:
4. depends: множество идентификаторов задач, от которых зависит текущая задача.
5. revdeps: множество идентификаторов задач, которые зависят от текущей задачи.
6. hash: хэш.
7. unihash: уникальный хэш, который используется для идентификации задачи.
8. task: имя задачи.
9. weight: вес задачи.
10. Выполнение задач в порядке очереди: BitBake выполняет задачи в порядке, определенном очередью выполнения, учитывая построенное дерево зависимостей.

## Структура проекта

Содержимое папки build:

- Папка build/tmp (система сборки OpenEmbedded создает и использует эту директорию для всех выходных данных системы сборки):

- build/tmp/hosttools - содержит хост-инструменты, которые необходимы для сборки проекта.

- build/tmp/pkgdata - содержит списки содержимого каждого пакета (RPM, DEB, IPKG и т. д.).

- build/tmp/buildstats - содержит статистику сборки, сгенерированная классом buildstats.

- build/tmp/cache/ - содержит кэш, полученный после анализа рецептов и файлов конфигурации. Он позволяет ускорять будущие сборки, если рецепты не были изменены.

- build/tmp/deploy/ - содержит результат выполнения процесса сборки OpenEmbedded.

- build/tmp/deploy/rpm - содержит .rpm пакеты, созданные процессом сборки. Пакеты рассортированы по папкам для разных типов архитектуры.

- build/tmp/deploy/licenses - содержит информацию о лицензировании пакетов.

- build/tmp/deploy/images - содержит объекты, сгенерированные в процессе сборки, включают в себя такие элементы, как образы загрузчика, ядра, корневые файловые системы и другое.

- build/tmp/deploy/spdx - хранит файлы, связанных с SPDX (Software Package Data Exchange), который является стандартом для описания метаданных программного обеспечения и лицензий.

- build/tmp/sstate-control/ - хранит файлы манифеста общего состояния. Манифесты отслеживают, какие файлы устанавливаются каждым этапом сборки ('sstate task'). Это позволяет системе определить, какие файлы могут быть удалены при очистке рецепта или когда устанавливается новая версия. Манифесты также используются для обнаружения перезаписи файлов одним этапом, если они были использованы другим этапом сборки.

- build/tmp/sysroots-components/ - компоненты sysroot, которые используются для подготовки среды выполнения для различных рецептов. Задача do\_prepare\_recipe\_sysroot использует содержимое этого каталога для создания sysroot для каждого рецепта в DEPENDS.

- build/tmp/sysroots-uninative/ - содержит часть корневой файловой системы (rootfs), которая включает в себя библиотеки, необходимые для работы сборочной системы и инструментов компилятора.

- build/tmp/stamps/ - хранит информацию, которую BitBake использует, чтобы отслеживать, какие задачи и когда выполнялись. Каталог подразделяется по архитектуре, имени пакета и версии.

- build/tmp/log/ - содержит общие логи.

- build/tmp/work/ - cодержит рабочие подкаталоги, в которых находится информация, специфичная для архитектуры, для пакетов, созданных BitBake. Все задачи выполняются из соответствующего рабочего каталога.

- build/tmp/work/{tunearch}/{recipename}/{version}/ - рабочий каталог рецепта. (${WORKDIR}). Его ключевые подкаталоги:

- ${WORKDIR}/temp - содержит логи каждой выполненной задачи, файлы «запуска» для каждой выполненной задачи, и файл log.task\_order, в котором указан порядок выполнения задач.

- ${WORKDIR}/image - содержит выходные данные задачи do\_install , соответствующие переменной ${D}.

- ${WORKDIR}/pseudo - содержит псевдобазу данных и данные о задачах, которые выполнялись с использованием псевдонима для определенного рецепта.

- ${WORKDIR}/sysroot-destdir - содержит выходные данные задачи do\_populate\_sysroot .

- ${WORKDIR}/package - содержит выходные данные задачи do\_package до того, как выходные данные будут разделены на отдельные пакеты.

- ${WORKDIR}/packages-split - содержит выходные данные задачи do\_package после разделения выходных данных на отдельные пакеты.

- ${WORKDIR}/recipe-sysroot - каталог, заполненный целевыми зависимостями рецепта. Этот каталог выглядит как целевая файловая система и содержит библиотеки, с которыми может потребоваться связать рецепт.

- ${WORKDIR}/recipe-sysroot-native - каталог, заполненный собственными зависимостями рецепта. Этот каталог содержит инструменты, необходимые для сборки рецепта (например, компилятор, Autoconf, libtool и т. д.).

- ${WORKDIR}/build - этот подкаталог применяется только к рецептам, поддерживающим сборки, в которых источник отделен от артефактов сборки. Система сборки OpenEmbedded использует этот каталог как отдельный каталог сборки (т. е. ${B} ).

- build/tmp/work-shared/ - для повышения эффективности система сборки OpenEmbedded создает и использует этот каталог для хранения рецептов, которые используют общий рабочий каталог с другими рецептами.

- Папка build/downloads:

Cодержит загруженные архивы с исходными кодами.

- Папка build/cache:

Cодержит несколько внутренних файлов, используемых системой сборки OpenEmbedded. Он также содержит sanity\_info - текстовый файл, в котором отслеживается важная информация о сборке, такая как значения TMPDIR , SSTATE\_DIR , а также имя и версия дистрибутива хоста.

- Папка build/conf:

Содержит файлы конфигурации.

- build/conf/local.conf - содержит все локальные пользовательские конфигурации для вашей среды сборки.

- build/conf/bblayers.conf - определяет слои, которые представляют собой деревья каталогов, по которым проходит BitBake.

- Папка build/sstate-cache/:

Этот каталог содержит общий кэш состояний.

## Переменные профилирования

В процессе сборки при использовании Yocto Project происходит логирование переменных нагрузки, тем самым собирается статистика сборки. Этими значениями можно оперировать для профилирования сборки:

1. Использованные ресурсы по каждой задаче:

По каждой задаче логируется список из 30 различных метрик ресурсов, самими информативными из которых являются:

1. Started – момент начала выполнения задачи
2. Ended – момент окончания выполнения задачи
3. Elapsed time - затраченное время
4. utime - количество времени, в течение которого процесс был запланирован в пользовательском режиме
5. stime - количество времени, в течение которого он был запланирован в режиме ядра
6. cutime - время, в течение которого дочерние процессы были запланированы в пользовательском режиме
7. cstime - время, в течение которого они были запланированы в режиме ядра
8. IO rchar - количество байт, считанных из хранилища
9. IO wchar - количество байт, записанных на диск
10. rusage ru\_maxrss максимальный размер резидентного набора в КБ (резидентный набор - то, сколько памяти выделено данному процессу и находится в оперативной памяти. Он включает в себя всю память, к которой процесс может получить доступ, включая память, которая выгружается, память, которая выделена, но не используется, и память из общих библиотек)
11. Нагрузка на процессор, оперативную память и дисковое пространство в каждый момент времени сборки

В процессе сборки отслеживаются и логируются нагрузки на процессор, оперативную память и дисковое пространство, наблюдения проводятся с частотой 1 секунда и фиксируются в файлах статистики.

1. Давление на процессор, оперативную память и I/O в каждый момент времени сборки

Помимо вышеуказанных величин, также фиксируются переменные давления на процессор, оперативную память и I/O. Аналогично, значения отслеживаются и записываются с частотой 1 секунда.

## Описание механизма Bitbake отслеживания свободного места в процессе сборки

В переменной BB\_DISKMON\_DIRS указаны директории, за свободным местом которых ведется отслеживание, также указаны минимальные значения для свободного пространства (minSpace) и действия (action), которые должны вызываться при отсутствии достаточного места. Примерно каждые 10 секунд bitbake получает информацию о текущем свободном месте директорий с помощью os.statvfs(), если это значение меньше чем minSpace, то выполняется действие action (STOPTASKS - прекращение выполнения новых задач и завершение выполнения текущих активных задач или HALT - немедленное прекращение выполнения всех задач).

## Контроль за значениями переменных нагрузки

Минимальное значение для каждой переменной - 1, максимальное -1000000. Если установленное значение меньше минимального, сборка сразу же прекращается, а если значение больше максимального, то установленное значение просто игнорируется.

Значения текущего давления берутся из файлов /proc/pressure/cpu, /proc/pressure/io, /proc/pressure/memory.

Хотя бы раз в секунду Bitbake сохраняет текущие значения давления (prev\_cpu\_pressure, prev\_io\_pressure, prev\_memory\_pressure), и каждый раз при выполении новой задачи происходит подсчет текущего давления. Если хоть одно значение превысило максимальное, то bitbake продолжает выполнять активные задачи, если на данный момент таковые были, и приостанавливает запуск новых задач, пока давление не упадет ниже установленного порога.

# Разработка архитектуры инструмента профилирования и анализа

## Docker-обёртка

Для удобной работы с проектом была создана Docker-обёртка, включающая скрипты со следующими опциями:

1. Проверка установки всех необходимых зависимостей
2. Создание образа среды для проекта
3. Получение доступа к файловой системе контейнера
4. Сборка образа Yocto
5. Запуск образа Yocto
6. Настройка логирования

## Приложение для обработки статистики

Для выявления узких мест в сборке и наиболее объемных задач было создано приложение для обработки статистики со следующими опциями:

1. Построение и анализ графа зависимостей с возможностью найти и отсортировать задержки между концом выполнения дочерней задачи и началом выполнения родительской задачи (для выявления возможного узкого места)
2. Ранжирование задач определенного типа по затратам определенного ресурса с возможностью указать тип задачи и ресурс, по которому будет проводиться ранжирование
3. Анализ нагрузки на процессор, оперативную память и I/O по всей временной линии сборки с возможностью найти временные интервалы, когда определенный ресурс не занят полностью, со списком задач, выполнявшихся в этот временной интервал.

## Приложение для отрисовки графиков нагрузки на определенный ресурс

Для возможности визуально отобразить и проанализировать затраченные ресурсы было реализовано приложение, визуализирующее затрату определенного ресурса на задачи. Пример графика (приложена часть графика):

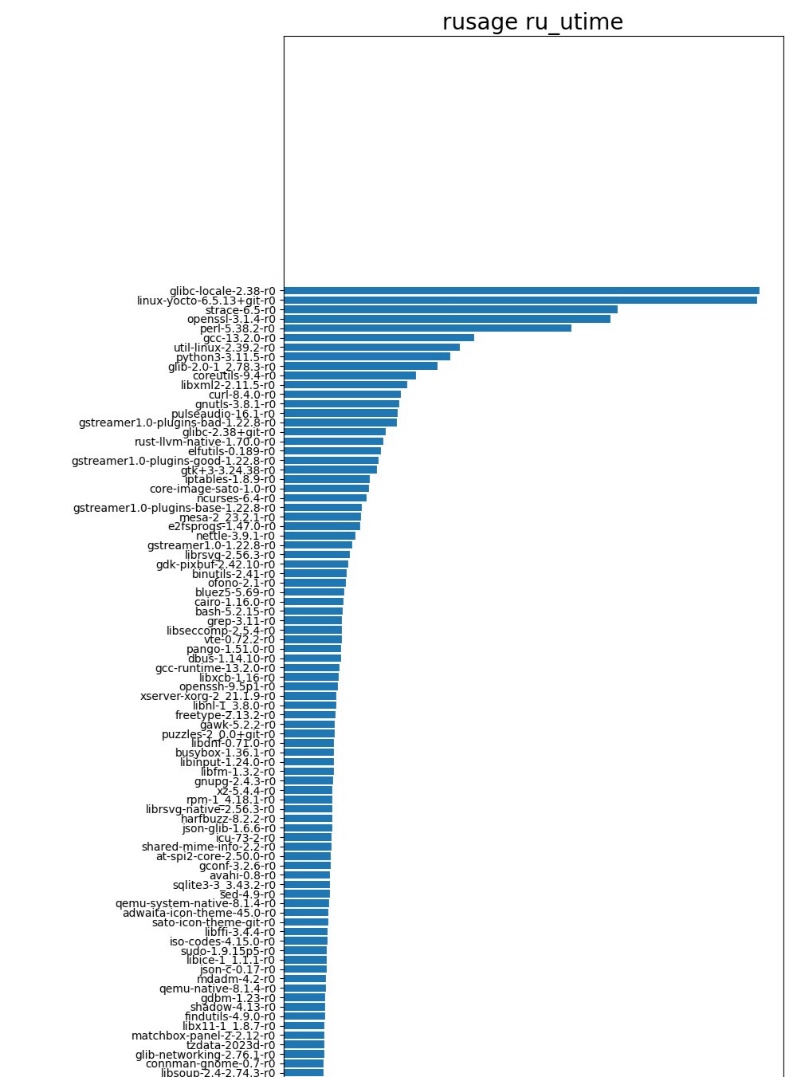


Рисунок 1. Пример графика нагрузки на ресурс rusage ru\_utime (часть)

# Профилирование

# Профилирование временной линии сборки при помощи встроенной утилиты Bitbake

При помощи встроенной утилиты Bitbake был построен временной график. Ось X данного графика соответствует времени сборки в секундах. Сверху находятся подграфики, отображающие

1. Нагрузку на процессор
2. Пропускную способность диска
3. Среднее CPU Pressure (это соотношение, измеряющееся в процентах, отслеживается как последние тенденции за десять, шестьдесят и триста секунд, что дает представление о краткосрочных событиях, а также о средне- и долгосрочных тенденциях - доля времени, в течение которой процессы на CPU останавливаются в ожидании "доступа" к CPU, информация берется из /proc/pressure/cpu)
4. Среднее I/O Pressure (то же самое, что и п.3, но в ожидании ввода-записи, информация берется из /proc/pressure/io)
5. Среднее MEM Pressure (то же самое, что и п.3, но в ожидании получения памяти, информация берется из /proc/pressure/memory)
6. Нагрузка на файловую систему (процент занятой/незанятой памяти)
7. Нагрузка на RAM (в том числе процент, занимаемый кэшем и буферами, а также график swap).

Далее идет временной график, отображающий выполнение каждой задачи. Для каждой задачи видно, когда она началась, когда закончилась, написано сколько секунд заняло выполнение, а также цветом обозначен тип задачи (конфигурирование/установка/компиляция и т.п.). Из графика можно понять, какие задачи выполнялись одновременно.

Так как график очень объемный, его можно посмотреть по ссылке: [[https://drive.google.com/drive/u/1/folders/1lh0FOV6EJJqjECu5wSaJJAor\_A9T2bgW](https://drive.google.com/drive/u/1/folders/1lh0FOV6EJJqjECu5wSaJJAor_A9T2bgW)]

## Визуализация графа зависимостей

Для лучшего понимания того, как происходит сборка, был визуализирован граф зависимостей. Сами данные графа в формате dot можно получить при помощи использования утилиты `bitbake -g <image>`.

Вершинами в графе зависимостей являются задачи. Задачи связаны ребром, если одна из них зависит от другой. Граф является ориентированным.

Была создана 2D-визуализация при помощи Python:

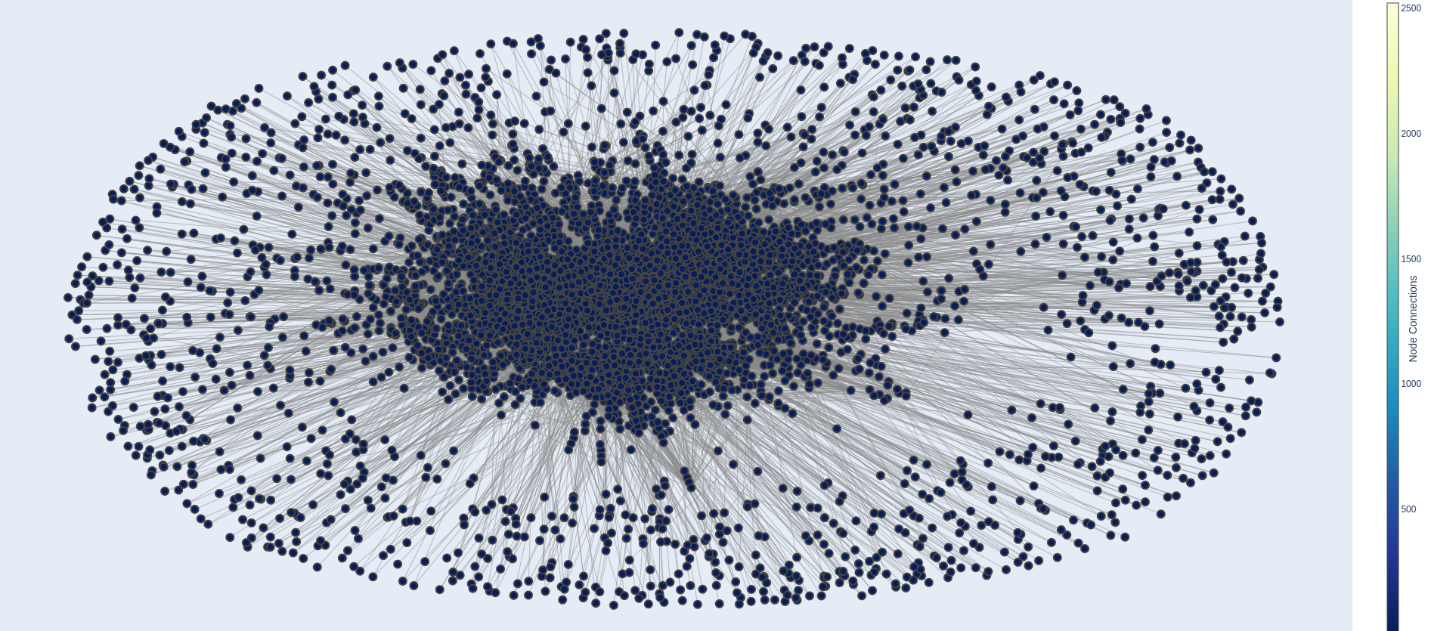


Рисунок 2. 2D-визуализация графа зависимостей с помощью python

Также была создана 3D-визуализация при помощи Graphia:

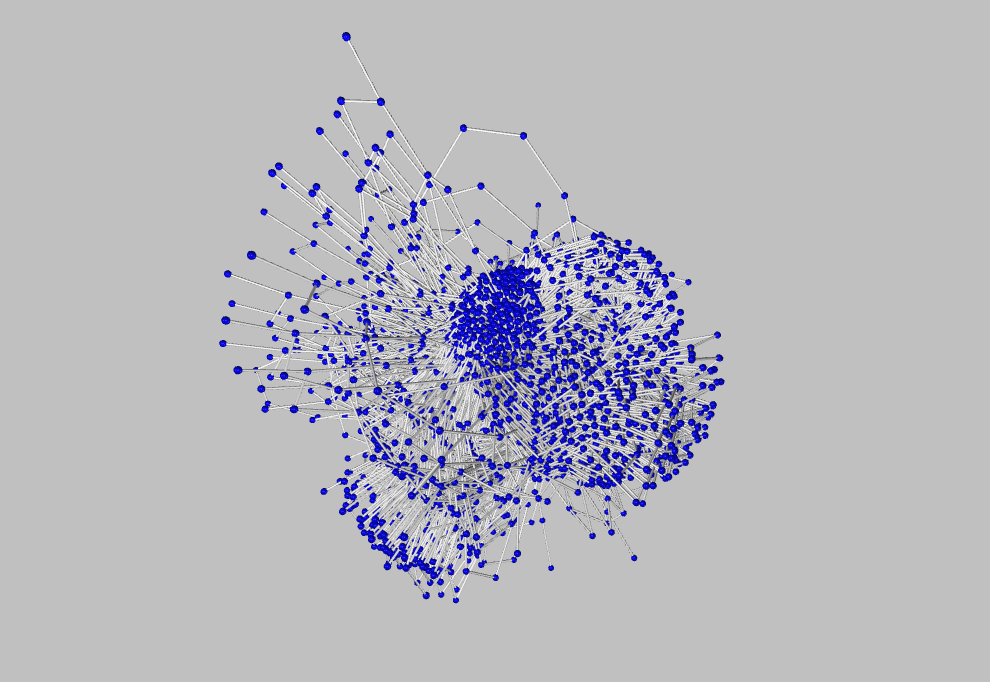


Рисунок 3. 3D-визуализация графа зависимостей при помощи Graphia

Также были визуализированы отдельные слои графа зависимостей. Под слоем понимается множество вершин, находящихся на одинаковой высоте от корневой вершины do\_build, а также вершины, напрямую связанные ребрами с вершинами этого множества. На визуализацию слоев можно посмотреть по ссылке: [<https://drive.google.com/drive/u/1/folders/1iDxxq7sxxZWLCpOhBl2Xb6UEQneB_4qW>]

## Изучение механизма пересборки образа

1. Было выявлено, что при наличии собранного образа и без внесения изменений в сборку пересборка не происходит - все необходимые данные выгружаются из кэша.
2. В то же время, если полностью удалить /build - будет полная пересборка без кэшируемых данных (поскольку кэш весь был в ./build, а именно в ./build/sstate-cache).
3. Также есть возможность отказаться от скачивания исходного кода, он располагается в папке ./build/downloads.

Был проведен эксперимент - два раза собран образ core\_image\_minimal. В первый раз сборка производилась без скачивания исходников, во второй раз была удалена папка /build/downloads - сборка производилась со скачиванием исходников. Оказалось, что сборка без скачивания исходников прошла на 17,5% быстрее.

Так как время сборки очень сильно зависит от наличия кэша, то разберем этот вопрос подробнее.

### Локальный кэш

1. Кэш процесса сборки располагается в папке poky/build/sstate-cache/
2. Кэш утилит сборки располагается в папке poky/build/cache/
3. Загруженный исходный код и бинарные файлы располагаются poky/build/downloads/

Есть возможность задать расположение локального кэша самостоятельно, изменяя значения переменных DL\_DIR (загруженный исходный код) и SSTATE\_DIR (кэш процесса сборки) в файле poky/build/conf/local.conf

### Конфигурация удаленной сборки и сборка

Можно сконфигурировать сборку таким образом, чтобы задачи загружались из кэша с удаленного зеркала. Это полезно, так как может существенно ускорить сборку на компьютере, где нет локального кэша.

Был проведен эксперимент с изменением конфигурации, в результате которого 99% задач загрузились с зеркала.

### Хэширование

Хэширование - использование хэш-функций для идентификации и проверки целостности сохраненных в кэше данных. Хэш-функция вычисляет уникальное хэш-значение для каждого объекта кэша, и это значение сохраняется вместе с артефактом в кэше. При повторном использовании артефакта из кэша хэш-функция снова вычисляет хэш-значение и сравнивает его с сохраненным значением, чтобы убедиться, что артефакт не был изменен или поврежден.

BitBake хэш-сервер - это сервер, который используется для хранения и предоставления хэш-значений для данных, созданных во время сборки BitBake. BitBake хэш-сервер используется для проверки целостности артефактов, предотвращения повторной сборки неизмененных артефактов и обеспечения надежности и безопасности сборок.

Рассмотрим две системы хэширования, которые можно настроить в Yocto Project: GPG и OEEquivHash. По умолчанию настроен OEEquivHash.

1. GPG

В Yocto Project для проверки пакетов может использоваться GPG-подпись. Когда пакет создается, он подписывается с помощью ключа GPG, и полученная подпись сохраняется вместе с пакетом. Когда пакет загружается, его подпись проверяется с помощью соответствующего открытого ключа GPG. Если подпись верна, то пакет считается подлинным, закрытый ключ же хранится где-то в качестве переменной сборки.

GPG (GNU Privacy Guard) - это свободная реализация OpenPGP (Pretty Good Privacy) стандарта, который предоставляет криптографические методы для обеспечения конфиденциальности, целостности и подлинности данных.

1. OEEquivHash

OEEquivHash отличается от GPG-подписи тем, что он не использует криптографические ключи для проверки подлинности пакетов. Вместо этого, он полагается на то, что хэш-суммы, сохраненные в файле с метаданными, были созданы доверенным источником. Его использования рекомендовано в условиях работы в закрытой или локальной сети, поскольку ускоряет хэш преобразования, упрощает само хэширование и работу с ним, ведь нет надобности администрировать еще и работу с ключами.

### Кэш-сервера

Кэш-сервер — это сервер, который используется для хранения и предоставления кэшированных данных.

В настоящее время продолжается работа над реализацией и анализом работы сборки при нескольких удаленных кэш-серверах.

# Заключение

В результате работы было проведено тщательное изучение технической литературы, касающейся Yocto Project и ее составляющих. Были изучены встроенные в Yocto инструменты для профилирования, а также была создана архитектура собственного инструмента профилирования, с помощью которых были построены графики и визуализации для анализа. Была реализована Docker-обёртка проекта, проведены эксперименты с различными конфигурациями сборок, изучены системы кэширования и хэширования, проведены эксперименты с локальной и удаленной пересборкой.