ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний день для автоматизации процессов сборки программного обеспечения применяются специализированные инструменты – системы сборки.

В общем случае система сборки – набор средств для преобразования данных из одного формата в другой. Чаще всего системы сборки используются для автоматизации таких задач как компиляция исходного кода в исполняемые файлы и создание программного пакета, предназначенного для конечного пользователя. Однако учитывая широкий спектр современных языков программирования и сред разработки, на систему сборки могут быть возложены различные задачи такие как запуск тестов или генерация документации.

Системы сборки, как правило, сложны для реализации и обслуживания. Плохо спроектированная система сборки может привести к тому, что разработчики тратят значительную часть времени на сборку. Согласно исследованию [1], даже в группах меньше 20 человек каждый разработчик тратит в среднем 12% времени на решение проблем, связанных со сборкой. При разработке больших проектов проблема усугубляется.

Проблемы, с которыми сталкиваются разработчики, включают:

- Ошибки компиляции из-за неправильной работы системы сборки – невозможно продолжить работу пока проблемы со сборкой не будут устранены. Чаще всего это решается переборкой всего проекта, что отнимает много времени.
- Некорректная сборка программы – в этом случае создается впечатление что в логике программы ошибка, тогда как проблема в процессе сборки.
- Медленная сборка – разработчики теряют время на ожидание окончания сборки.
- Сложность внесения изменений в систему сборки – если процесс сборки сложен для понимания, то внесение изменений в логику работы системы сборки занимает много времени.

Целью работы является исследование существующих систем сборки и разработка системы сборки, демонстрирующей приемы для ускорения процесса сборки.

Для достижения поставленной цели в рамках данной работы решаются следующие задачи:

- Выполнить аналитический обзор подходов к сборке программного обеспечения.
- Исследовать существующие системы сборки, определить их положительные стороны и недостатки.
- Реализовать систему сборки и продемонстрировать ее работу.
- Сделать выводы по проделанной работе.

Практическая ценность данной работы заключается в том, что разработанную систему сборки можно использовать для сборки программного обеспечения. Работа также может быть использована как руководство для выбора, разработки, и использования систем сборки.

ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Задачи системы сборки

В разработке программного обеспечения чаще всего встречаются следующие сценарии сборки [2]:

- Компиляция ПО, написанного на компилируемых языках программирования таких как C, C++ и Java (Рисунок 1).
- Упаковка ПО, написанного на интерпретируемых языках таких Perl и Python, исполняемых файлов и конфигурационных файлов в конечный программный пакет.
- Генерация и последующая компиляция исходного кода на основе шаблонов.
- Выполнение модульных тестов для проверки небольших частей программного обеспечения в изоляция от остальной части кода.
- Выполнение инструментов статического анализа для выявления ошибок в исходном коде программы. Выводом в данном случае является отчет об ошибках, а не исполняемый файл.
- Генерация документации в удобном для чтения формате таком как HTML или man страницы.

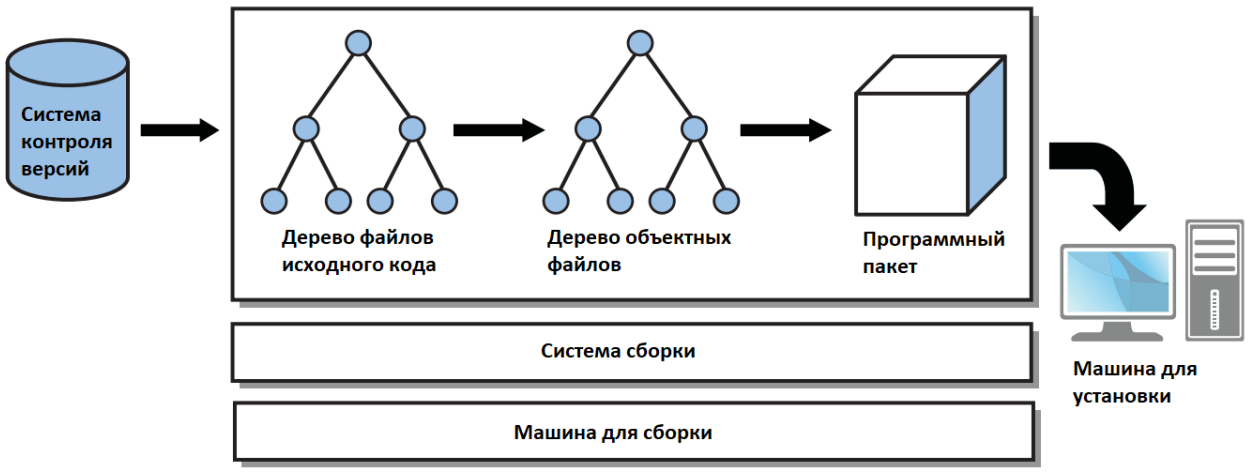

Рисунок 1 – обобщенный сценарий сборки для компилируемых языков

1.2 Требования к системам сборки

Система сборки должна удовлетворять четырем требованиям [3]: она должна быть удобной для использования, быстрой, корректной, и масштабируемой.

1.2.1 Удобство использования

Полезность системы сборки в первую очередь определяется тем как хорошо она помогает разработчику достичь свои цели. Удобство – это широкая область, которая зависит от множества факторов таких как поддержка желаемой функциональности, понятность пользовательского интерфейса и прозрачность работы.

1.2.2 Корректность работы

Если система сборки не может гарантировать правильное поведение во всех сценариях использования, разработчик потеряет к ней доверие, что приведет к увеличению времени сборки. Например, объектные файлы зависят не только от соответствующих файлов исходного кода, но и от используемых заголовочных файлов. Если система сборки это не учитывает, то изменение используемого заголовочного файла не приведет к перекомпиляции файла исходного кода. Заметив, что система сборки не работает должным образом, разработчик чаще всего удаляет все созданные ранее объектные и исполняемые файлы, чтобы «убедить» систему сборки выполнить компиляцию.

1.2.3 Быстрота сборки

Во время сборки большая часть вычислительной мощности тратится на инструменты, вызванные системой сборки. Поэтому можно добиться наибольшего ускорения сборки минимизируя эти вызовы. Требование быстроты сборки в

некотором роде противоречит требованию корректности, поскольку более сложные методы обеспечения корректности вызывают большие накладные расходы по памяти и вычислениям.

1.2.4 Масштабируемость

Программные системы постоянно растут по размеру и сложности. Возникает необходимость поддерживать новые платформы. В сборке задействуется множество различных инструментов. Масштабируемая система сборки может справляться с очень большими программными системами и адаптироваться к новым требованиям.

1.3 Особенности систем сборки

1.3.1 Частичная сборка

Большие программные проекты содержат большое количество модулей, программ, обязательных и дополнительных частей. Пользователь должен иметь возможность выбрать часть проекта, который должен быть собран. Это могут быть каталоги, объектные файлы, программы, отдельные модули или их наборы.

1.3.2 Поддержка различных конфигураций

Поддержка различных конфигураций означает возможность создавать разные версии результата сборки из одних и тех же исходных файлов. Примером является отладочная конфигурация, при использовании которой в результат сборки включается дополнительная информация для отладки, и оптимизированная конфигурация, использующаяся в версии продукта для конечного пользователя. Другим распространенным случаем является сборка под различные архитектуры.

1.3.3 Репозитории

Программные проекты обычно находятся в репозитории системы контроля версий такой как git или svn. Система сборки должна это учитывать и размещать артефакты сборки таким образом, чтобы разработчик не мог случайно добавить их в систему контроля версий.

1.3.4 Кэширование результатов компиляции

Во время сборки создается множество промежуточных файлов. Можно сэкономить время переиспользуя эти файлы во время сборки или между сборками. Например, объектный файл может использоваться во время линковки нескольких исполняемых файлов. В этом случае если файл закэширован во время сборки первого исполняемого файла, то нет необходимости компилировать его снова в дальнейшем. Необходимо следить за тем, чтобы кэшированные файлы использовались только если команды компиляции совпадают с теми, которые использовались во время добавления файла в кэш. Чтобы добавить поддержку кэширования используются такие утилиты как ccache [5], использующиеся в качестве обертки над компилятором.

1.3.5 Параллельная сборка

Многоядерные и многопроцессорные архитектуры широко распространены и должны быть использованы для ускорения процесса разработки. Система сборки должна иметь возможность планировать параллельное исполнение отдельных этапов, не нарушая зависимости между ними.

Закон Амдала [6] гласит, что максимальное ускорение программы ограничено временем выполнения ее последовательных инструкций:

$$S_n = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

$S_n -$ во сколько раз можно ускорить вычисления (ускорение)

$n -$ количество процессоров (ядер)

$\alpha -$ доля последовательно исполняемых вычислений

Это стоит принимать во внимание, так как в большинстве случаев не все этапы сборки можно выполнять параллельно.

1.3.6 Распределенная компиляция

Идея распределенной компиляции проста. Несколько машин, предназначенных для распределенной компиляции, объединяются в одну сеть. Машины настраиваются выполнения этапов сборки, и одна из них распределяет этапы сборки между другими. Для того чтобы получить выгоду от распределенной компиляции, распределение этапов сборки и пересылка необходимых данных должны быть быстрее чем локальная сборка. Поэтому целесообразность использования распределённой сборки прежде всего зависит от инфраструктуры.

Утилита distcc [7] может использоваться как обертка над вызовом компилятора, и хорошо интегрируется в системы сборки. Вывод препроцессора отправляется на сервера сборки для компиляции, после чего получившиеся объектные файлы отправляются обратно. Одна из проблем данного подхода – различия в среде сборки (например, разные версии компиляторов) на серверах сборки могут привести к непредсказуемым ошибкам.

ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ ИНСТРУМЕНТОВ СБОРКИ

2.1 Make

Утилита Make – прабабушка всех систем сборки. Созданная в 1997 году, она до сих пор остается очень популярным инструментом [8]. Правила сборки описываются в специальном файле Makefile в формате:

```
1 цель: зависимости
2     действия
```

Утилита поддерживается на многих платформах, но правила сборки для каждой платформы разработчику необходимо писать самому.

2.2 Ant

Ant – утилита, предназначенная для сборки проектов на Java, создана в 2000 году как аналог Make [9]. В отличие от Make, Ant предоставляет библиотеку готовых к использованию правил сборки.

2.3 Maven

Maven – система сборки Java проектов [10]. В отличие от Ant и Make, в проектах на Maven не описываются действия для сборки проектов. Вместо этого описывается конфигурация проекта (название, используемые файлы исходного кода, внешние библиотеки, и т.д.) в соответствии с принятыми условными соглашениями.

2.4 CMake

CMake – система сборки C/C++ проектов высокого уровня [11]. Непосредственно сборкой она не занимается, а только генерирует инструкции

сборки для инструментов более низкого уровня, таких как Make (Рисунок 2). В файле CMakeLists.txt содержится платформо-независимое описание проекта: тип проекта (библиотека или исполняемый файл), используемые файлы исходного кода, подключаемые библиотеки, и т.д.

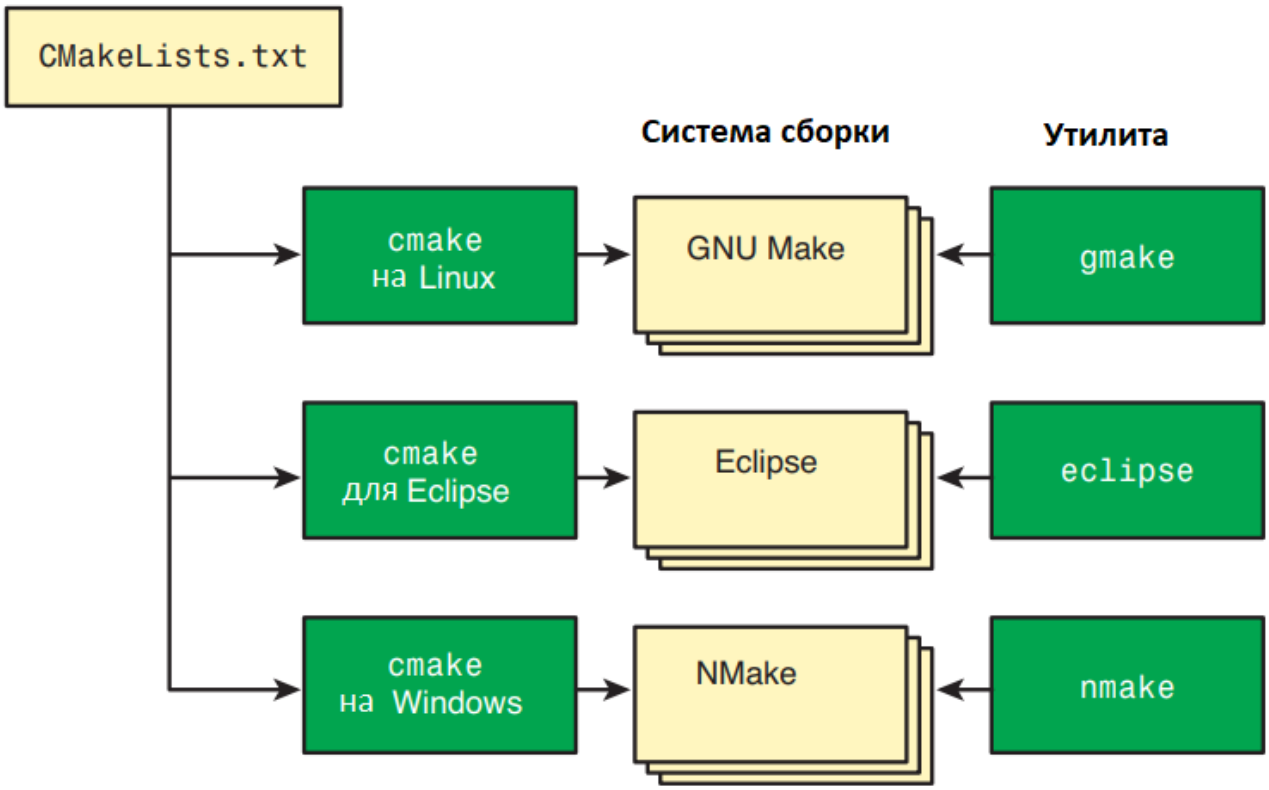

Рисунок 2 – использование CMake на различных платформах

2.5 Gradle

Gradle – система сборки проектов на Java и C/C++, использующая для описания проектов предметно-ориентированный язык, основанный на Groovy [12]. Отличительной особенностью Gradle является скорость частичной сборки Java проектов по сравнению с Maven.

2.6 Bazel

Bazel – система сборки проектов на Java и C/C++ [13]. Изначально была разработана Google для собственных нужд и позже открыта для общего пользования. Отличительной особенностью является жёсткая структура описания проектов, обеспечивающая хорошую масштабируемость сборки для больших проектов.

2.7 Сравнение и выводы

Рассмотренные системы сборки предоставляют различную функциональность (Таблица 1).

Таблица 1 – сравнение функциональности систем сборки

Функциональность	**Make**	**Ant**	**Maven**	**CMake**	**Gradle**	**Bazel**
Поддержка разных платформ (Linux, OS X, Windows)	+	+	+	+	+	+
Сборка C/C++				+	+	+
Сборка Java		+	+		+	+
Кэширование результатов компиляции						
Параллельная сборка	+	+	+	+	+	+
Распределенная сборка						
Модульное тестирование		+	+	+	+	+
Работа с удаленными репозиториями			+		+	+
Предварительно скомпилированные заголовки					+	
Создание программного пакета						

Современные инструменты сборки такие как Bazel и Gradle предоставляют возможность работать с большими и сложными программными проектами, но при этом сами являются таковыми. В случаях, когда для сборки программного проекта требуется функциональность, не предоставляемая существующими решения, часто создаются специализированные системы сборки, удовлетворяющие требованиям конкретного проекта. Например, из Таблицы 1 видно, что механизмы кэширования результатов компиляции, распределенной сборки и создания программного пакета не реализованы ни в одной из рассмотренных систем сборки.

ГЛАВА 3. РАЗРАБОТКА СИСТЕМЫ СБОРКИ

3.1 Формулировка задач системы сборки

Исследование существующих систем сборки показало, что рассмотренные инструменты не предоставляют следующие механизмы, полезные при разработке C/C++ проектов:

1. Кэширование результатов компиляции
2. Механизм распределенной сборки
3. Механизм версионирования результатов сборки
4. Создание программных пакетов на основе собранных проектов
5. Создание пакета обновлений.

Реализация этих механизмов позволит ускорить процесс сборки и автоматизировать создание программного пакета для конечного пользователя.

Помимо перечисленных выше пунктов, разработанная в рамках данной работы система сборки также должна предоставлять основную функциональность для сборки, реализованную в современных системах сборки:

1. Поддерживать модульную сборку проектов C/C++ на платформе Linux и предоставлять механизм обеспечения зависимостей между проектами
2. Обеспечивать механизм параллельной сборки
3. Поддерживать запуск тестов во время сборки
4. Поддерживать использование предварительно скомпилированных заголовков

Для разработки системы сборки были использованы язык C и Make.

3.2 Модульная сборка проектов

Для описания пользовательского проекта был выбран формат Makefile, в котором указываются основные характеристики проекта:

- Тип и название
- Используемые файлы исходного кода
- Зависимости на другие проекты

Такой формат позволяет достаточно просто описать проект с помощью знакомого многим разработчикам синтаксиса Make. Пример описания пользовательской библиотеки:

```
1 LIBRARIES = market_data
2
3 SOURCES  = $(wildcard src/*.cpp)
4 SOURCES += $(wildcard src/server/*/*.cpp)
5 SOURCES += $(wildcard src/market_data_protocol/*.cpp)
6
7 DEPENDS += libraries/util
8 DEPENDS += libraries/error
9 DEPENDS += libraries/protocol_impl
10  DEPENDS += libraries/storage
11  DEPENDS += libraries/subscription
12
13  include $(MK_BUILD_SYSTEM)/main.make
```

Для того использовать функциональность системы сборки в конце файла подключается файл системы сборки main.make. Так как Makefile проекта может располагаться на произвольном уровне дерева проектов на диске, необходимо динамически определить путь до файла системы сборки. Для этого на языке C была разработана утилита tbmake. Утилита определяет корень дерева файлов системы сборки, проверяя находится ли каталог системы сборки на одном из уровней выше текущего рабочего каталога. Найденный путь передается в качестве аргумента

(MK_BUILD_SYSTEM) утилите Make для сборки проекта. Таким образом, пользователю достаточно вызвать tbmake из каталога проекта, и не нужно вручную указывать местонахождение системы сборки на диске.

Каждый проект указывает необходимые зависимости на другие проекты. Чтобы задать направленный граф всех зависимостей (Рисунок 3), был реализован рекурсивный алгоритм обхода зависимостей проектов в глубину, где на каждой итерации система сборки считывает зависимости проекта из Makefile в список смежности.

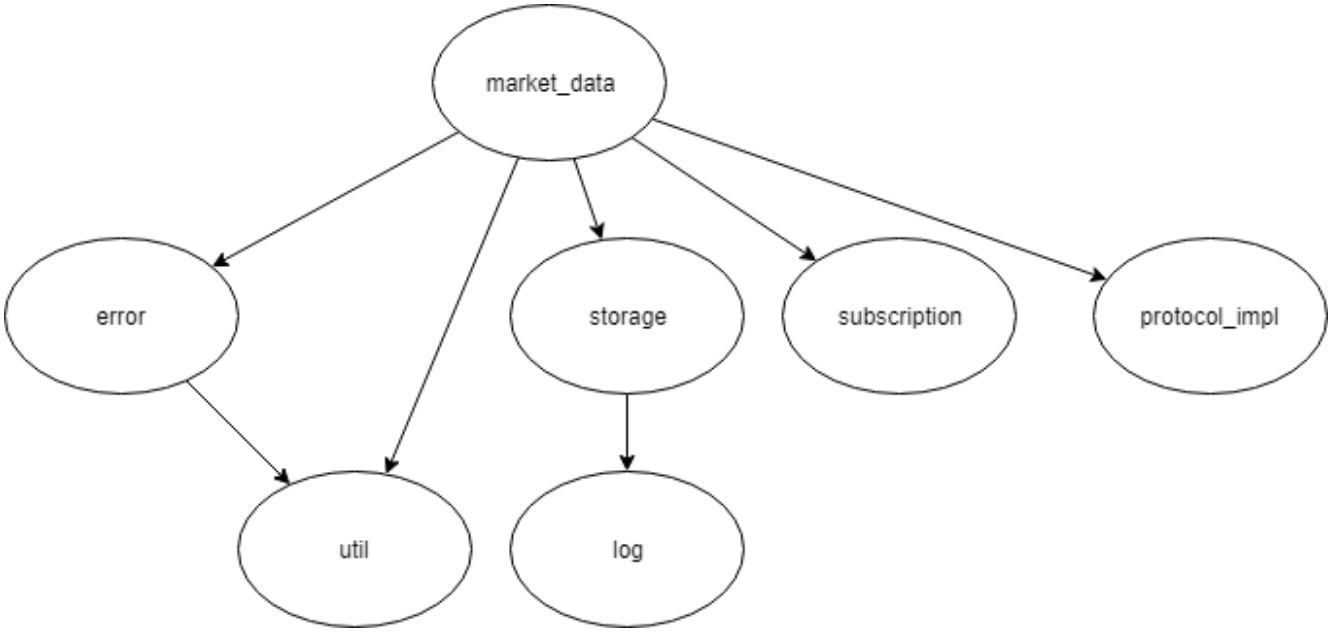

Рисунок 3 – граф зависимостей проекта market_data

После того как граф зависимостей составлен, можно начать сборку проектов. Для этого был реализован алгоритм обратного (англ. – postorder) прохода в глубину по графу зависимостей, при котором зависимости каждого проекта собираются до самого проекта:

```
$ tbmake
--- [tbricks_util] Build Started
--- [tbricks_util] Build Finished
--- [tbricks_error] Build Started
--- [tbricks_error] Build Finished
--- [tbricks_log] Build Started
```

```
--- [tbricks_log] Build Finished
--- [tbricks_protocol] Build Started
--- [tbricks_protocol] Build Finished
--- [tbricks_subscription] Build Started
--- [tbricks_subscription] Build Finished
--- [tbricks_storage] Build Started
--- [tbricks_storage] Build Finished
--- [tbricks_market_data] Build Started
--- [tbricks_market_data] Build Finished
```

3.3 Модульное тестирование во время сборки

Хотя создание хорошего набора тестов предполагает значительную работу, преимущества велики. Модульное тестирование во время сборки позволяет отловить возможные проблемы на ранней стадии разработки. Выполнение всех тестов может занять некоторое время, но подробный отчет о том, где именно лежит проблема, намного удобнее чем отладка полной исполняемой программы.

Во многих средах разработки стандартной практикой является считать программное обеспечение сломанным, если модульные тесты не проходят на 100%. Если происходят какие-либо сбои, считается, что код не готов к совместному использованию другими разработчиками.

В рамках работы был разработан универсальный механизм исполнения тестов, основанный на возможности модульной сборки проектов в разработанной системе сборки. С точки зрения системы сборки тестом является любой проект исполняемого файла в каталоге с названием test. Отличие от обычного проекта исполняемого файла в том, что собранный исполняемый файл исполняется после сборки. Такой подход обеспечивает возможность работы с любыми прикладными программными платформами тестирования, в отличие от существующих систем сборки (например, Gradle), в которых поддерживается работа только с ограниченным набором таких платформ (например, GoogleTest).

Выполнение всех тестов может занимать некоторое время. Поэтому целесообразно выполнять тесты только если были совершены изменения в тестируемый проект. Однако, если тест не выполнился успешно, необходимо продолжить исполнять тест в последующих запусках сборки, иначе разработчик может не заметить или забыть, что что-то пошло нет так. Для этого был реализован следующий механизм: при успешном завершении теста система сборки создает специальную файл-метку, а если тест не завершился успешно, то метка должна быть удалена. Тогда, если метка отсутствует, значит тест необходимо запустить. Однако следует учитывать, что работа системы сборки может быть экстренно прервана в любой момент (в том числе во время исполнения теста), поэтому целесообразно безусловно удалять метку непосредственно перед исполнением теста, а не после.

```
1 ifeq ($(wildcard $(TEST_TIMESTAMP)),)
2 RUN_TEST := YES
3 endif
4
5 run_test:
6   $(RM) $(TEST_TIMESTAMP)
7   $(ECHO) "--- Execute"
8   $(TEST_BINARY) $(TEST_FLAGS)
9   $(ECHO) "--- Passed"
10      $(MKDIR) $(TEST_TIMESTAMP)
```

Пример исполнения теста во время сборки представлен в разделе Приложение 1.

3.4 Параллельная сборка

Система сборки должна поддерживать как параллельную компиляцию, так и параллельную сборку проектов в целом, учитывая зависимости между проектами.

Сборка одного проекта не всегда задействует все ресурсы процессора (например, когда выполняется линковка в один поток). Соответственно сборка нескольких независимых между собой проектов позволяет ускорить общую сборку.

Для того чтобы определить влияние параллельной компиляции на время сборки были проведены следующие эксперименты: сборка проекта в один поток и сборка проекта в 8 потоков (Рисунок 4).

Эксперименты по сборке проводились на реальном проекте со следующими параметрами:

- Язык: C++
- Количество файлов: 665
- Количество строк кода: 163107

Эксперименты проводились с помощью утилиты time на тестовом стенде со следующими параметрами:

- Операционная система: Red Hat Enterprise Linux 6.8
- Процессор: Intel Xeon CPU E5-2666 v3 @ 2.90 GHz
- Количество ядер процессора: 10
- Оперативная память: 30 Гбайт
- Компилятор: clang 4.0

Данный тестовый стенд использовался для всех последующих экспериментов.

Сборка в 8 потоков:

real 4m11.597s
user 31m20.341s
sys 1m40.165s

Сборка в 1 поток:

real 32m5.977s
user 30m24.127s

sys 1m38.295s

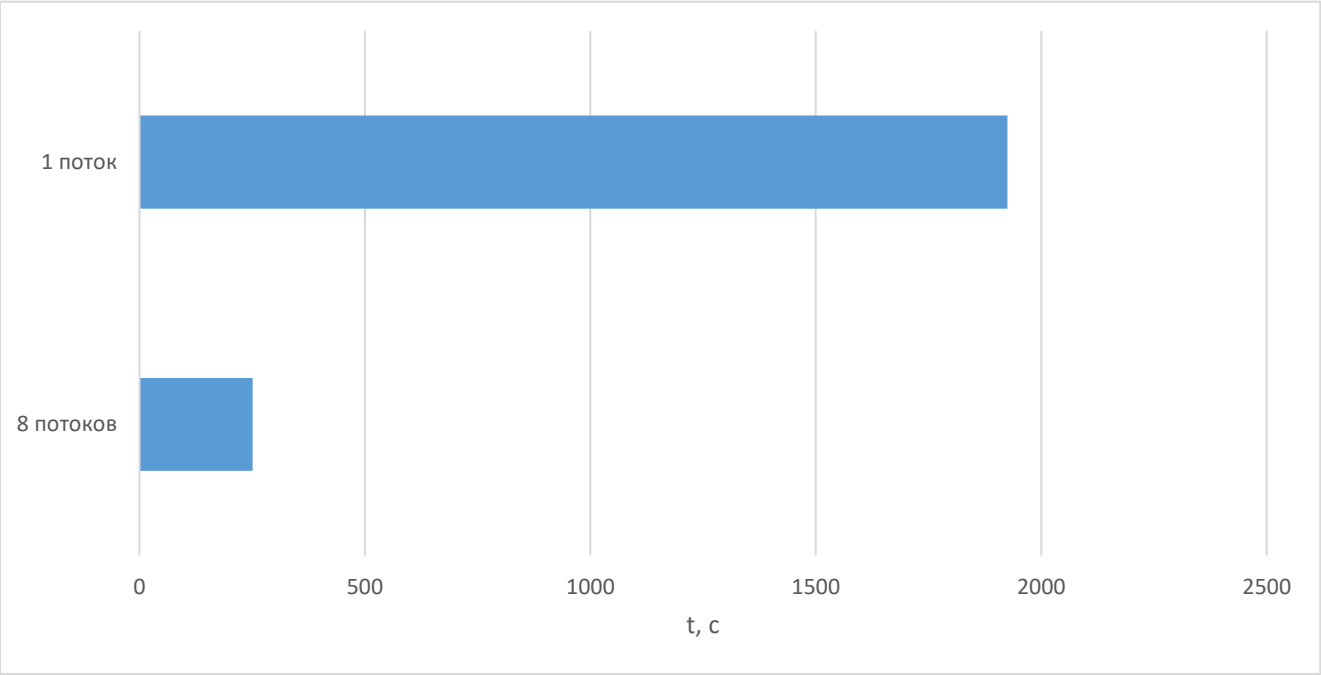

Рисунок 4 – сравнение длительности сборки (1 и 8 потоков)

Как и следовало ожидать, сборка в 8 потоков быстрее чуть менее чем в 8 раз по сравнению со сборкой в один поток за счет того, что некоторые действия (линковка) выполняются только в один поток.

3.5 Распределенная сборка

Чтобы реализовать распределенную сборку C/C++ проектов, в систему сборки была интегрирована утилита distcc. Утилита используется как обертка над компилятором:

```
1 ifeq ($(DISTCC),YES)
2     CC  := distcc $(CC)
3     CXX := distcc $(CXX)
4 endif
```

Переменные CC и CXX – компиляторы языка C и C++ соответственно.

Для распределённой сборки необходимо иметь список удаленных машин, на которых можно проводить компиляцию. Чтобы найти все доступные машины в подсети был разработан следующий скрипт:

```
1 echo -n "--randomize localhost/$N_LOCAL_THREADS " >
  ~/.distcc/hosts
2 nmap 10.201.1.0/25 --open -p 3632 \
3 | grep 'Nmap scan report for' \
4 | sed 's/Nmap scan report for //' \
5 | sed "s/$/\/$N_REMOTE_THREADS/" \
6 | grep -v `hostname -f` | sort -R \
7 | tr '\n' ' ' >> ~/.distcc/hosts
```

Где:

$N_LOCAL_THREADS – количество используемых потоков на локальной машине.

$N_REMOTE_THREADS – количество используемых потоков на каждой удаленной машине.

Для того чтобы определить влияние параллельной сборки на время сборки по сравнению с распределенной сборкой были проведены следующие эксперименты: сборка проекта в 8 потоков на локальной машине и сборка проекта в 8 потоков на 4 удаленных машинах, каждая из которых предоставляет 2 потока для сборки (Рисунок 5).

Все машины находятся в 10Gbps сети в одном датацентре.

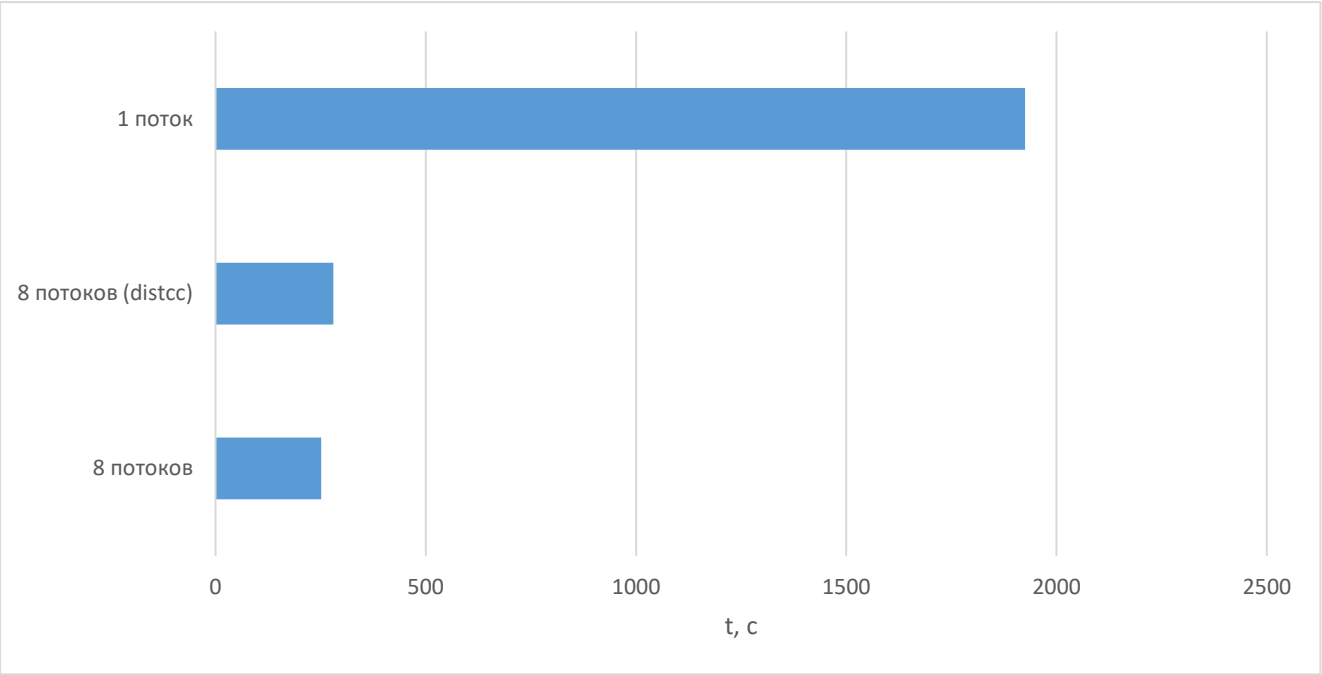

Рисунок 5 – сравнение длительности распределенной и локальной сборки

Как видно из графика, сборка с использованием distcc почти не уступает сборке на локальной машине по времени сборки. Небольшая разница во времени сборки объясняется накладными расходами на отправку данных на удаленные машины и получение данных обратно. Также роль играет то что препроцессор запускается для каждого файла с исходным кодом локально.

3.6 Кэширование результатов компиляции

Для реализации кэширования результатов компиляции в систему сборки была интегрирована утилита ccache. В общем случае, ccache используется как обертка над вызовом компилятора, сохраняя результат компиляции в отдельный каталог, или используя уже закэшированный результат компиляции вместо вызова компилятора. Чтобы определить, можно ли использовать кэш для данной команды компиляции, ccache считает хэш сумму файла исходного кода (а также используемых заголовков), команды компиляции, и информации о компиляторе.

Реализованный механизм кэширования результатов компиляции многократно ускоряет сборку в следующих случаях:

1. Разработчик сменил ветку в системе контроля версий и выполнил сборку. В этом случае объектные файлы, созданные во время сборки исходной ветки, заменяются новыми. Когда разработчик вернется на исходную ветку объектные файлы можно будет взять из кэша.
2. Разработчик создал новую копию проекта на диске и выполнил там сборку. Такая ситуация часто возникает, когда из-за большой длительности сборки разработчики создают копию проекта вместо того чтобы переключить ветку в текущем дереве проекта. Если результаты компиляции не зависят от местонахождения проекта на диске, также используются закэшированные результаты компиляции.
3. На сервере непрерывной интеграции выполняется сборка множества веток системы контроля версий. В большинстве случаев новые ветки базируются на ограниченном количестве долгоживущих веток (например, master или development). Первая сборка новых веток ускоряется за счет кэширования.

Следует учитывать, что в некоторых случаях в скомпилированных объектных файлах могут присутствовать абсолютные пути, что препятствует использованию кэша в разных каталогах. Например, при использовании ключа компилятора -g остается отладочная информация:

```
$ objdump -g StringUtils.cpp.1.o | grep DW_AT_comp_dir
DW_AT_comp_dir      DW_FORM_strp
<1a>   DW_AT_comp_dir    : (indirect string, offset: 0x41):
/export/home/fedor.kalugin/tb/worktree_1/src/libraries/cpph
elper
```

Для решения этой проблемы в разработанной системе сборки использован ключ компилятора -fdebug-prefix-map, заменяющий абсолютный путь на относительный:

```
CXXFLAGS += -fdebug-prefix-map=$(PROJECT_DIR)=./
```

Отладочная информация с использованием -fdebug-prefix-map:

```
$ objdump -g StringUtils.cpp.1.o | grep DW_AT_comp_dir
DW_AT_comp_dir      DW_FORM_strp
<1a>   DW_AT_comp_dir    : (indirect string, offset: 0x41):
./src/libraries/cpphelper
```

Для того чтобы определить влияние кэширования на время повторной сборки были проведены следующие эксперименты: сборка проекта в 8 потоков с предварительно очищенным кэшем, и повторная сборка проекта в 8 потоков, используя кэш (Рисунок 6).

```
$ time tbmake -sj8 CCACHE=YES
cache hit (direct)                    676
cache hit (preprocessed)                0
cache miss                              0
cache hit rate                     100.00 %
called for link                         1
real    0m5.384s
user    0m20.094s
sys     0m7.855s
```

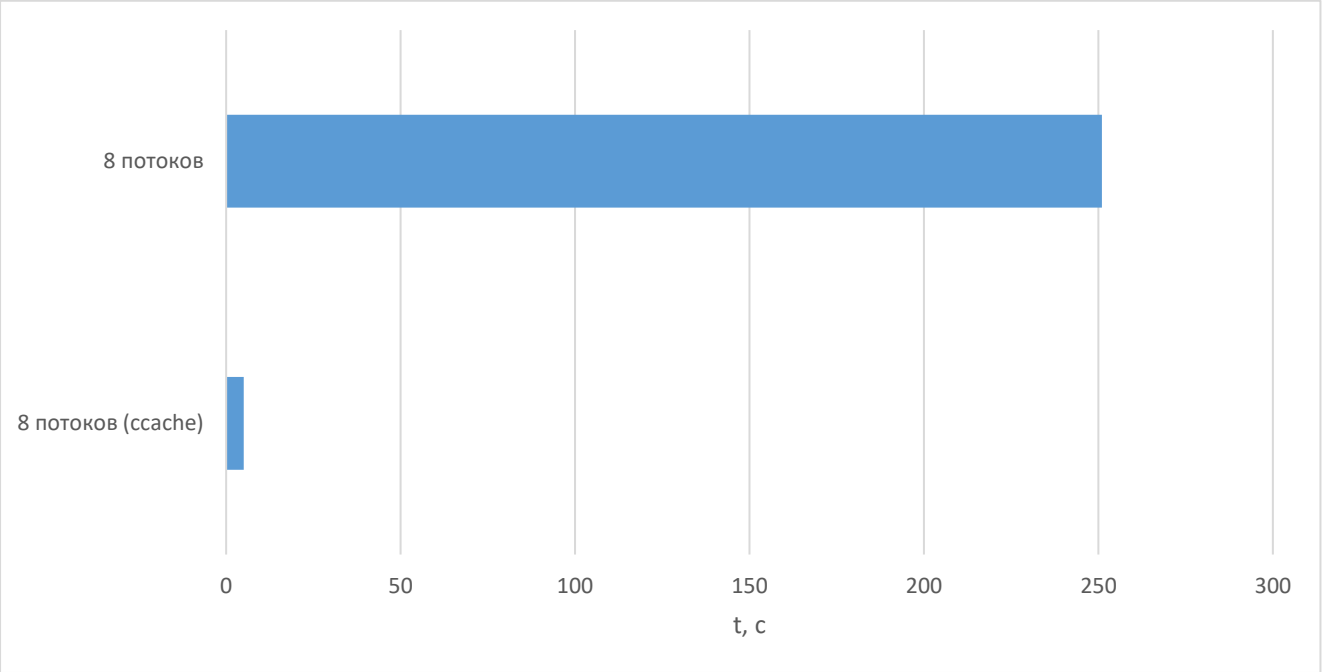

Рисунок 6 – сравнение длительности первичной и повторной сборки (ccache)

3.7 Предварительно скомпилированные заголовки

Чтобы сократить время компиляции были разработаны предкомпилированные заголовки. Поскольку заголовочные файлы меняются гораздо реже файлов, содержащих код программы (а библиотечные — практически никогда), разумным средством оптимизации является выполнять предварительную обработку заголовков, и преобразование их в файлы специального вида, которые при компиляции программы можно подключать, минуя первые стадии компиляции (создание абстрактного синтаксического дерева). За счёт предкомпиляции заголовков полной обработке компилятором подвергаются только изменённые части программы. Однако, предварительная компиляция заголовка помогает не всегда:

- При изменении любого из предкомпилируемых заголовков перекомпилируется весь набор.
- При полной перекомпиляции выигрыш по времени получается, когда один и тот же набор применяется как минимум в двух единицах компиляции.

Поэтому, как правило, в предкомпилируемый набор включают всевозможные библиотечные заголовки, крупные и в то же время редко изменяющиеся.

Чтобы скомпилировать заголовок система сборки использует флаги компилятора:

```
-x c++ -emit-ast
```

Система сборки затем подключает его с помощью флага:

```
-include-pch $(PCH_FILE)
```

Для того чтобы определить влияние предварительно скомпилированных заголовков на время сборки были проведены следующие эксперименты: сборка проекта в 8 потоков, где файлы с исходным кодом напрямую включают необходимые заголовки, и сборка проекта, где сначала компилируется и затем

используется общий заголовок (Рисунок 7). Содержимое заголовка представлено в разделе Приложение 2.

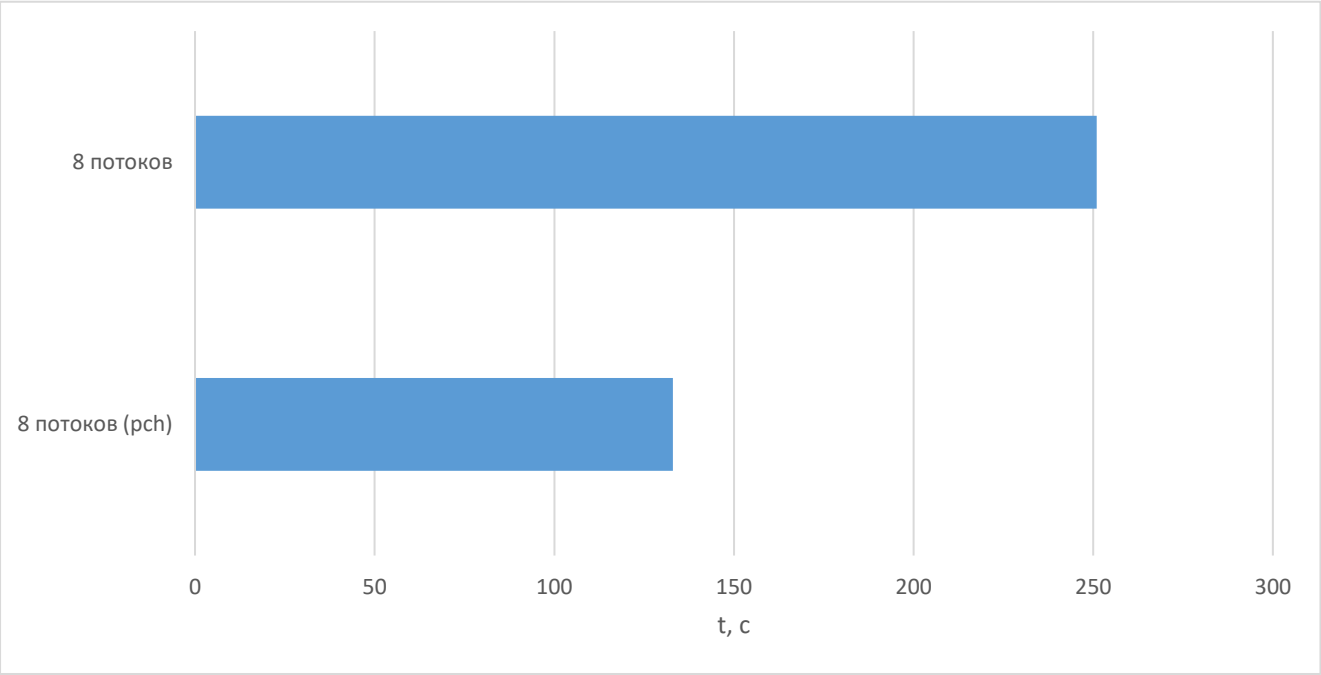

Рисунок 7 – сравнение длительности сборки

Как видно из графика использование предкомпилированных заголовков позволяет ускорить сборку проекта почти в 2 раза.

3.8 Версионирование результатов сборки

Во время отладки версия продукта и ревизия системы контроля версий, на которой был собран бинарный исполняемый файл, может быть не известна заранее. В таких случаях возникает необходимость сохранять информацию о версии внутри бинарного файла. Система сборки может осуществить это следующим образом: во время сборки генерируется файл исходного кода (на языке C/C++), содержащий строку с информацией о версии:

```
static const char __tbricks_version[] = "Tbricks-2.12.0
(235e628892e)";
```

Этот файл компилируется и используется во время линковки вместе с остальными объектными файлами. Чтобы узнать версию бинарного файла разработчику достаточно выполнить следующую команду:

```
$ strings mybinary.so | grep Tbricks
Tbricks-2.12.0 (235e628892e)
```

Система сборки должна учитывать, что ревизия системы контроля версий может меняться достаточно часто, и это не должно приводить к пересборке проекта. Однако если проект был изменен, то во время линковки должна быть использована новая версия. Чтобы это реализовать генерация объектного файла с версией была вынесена в отдельный проект, который добавляется в зависимости остальным проектам:

```
DEPENDS += share/version
```

Если у проекта есть эта зависимость, сгенерированный объектный файл добавляется в список:

```
MK_ADDITIONAL_OBJECTS += $(MK_BUILD_INC)/version/version.o
```

3.9 Создание программного пакета

В работе был реализован механизм создания программного пакета в виде архива, который содержит требуемые исполняемые программы, библиотеки и файлы данных, находящиеся в дереве сборки на диске. Так же, как и для отдельных C/C++ проектов, для описания программного пакета был выбран формат Makefile. Это позволяет разработчикам описывать программный пакет в привычном формате и иметь несколько проектов для создания разных программных пакетов. В проекте указываются отдельные файлы и другие C/C++ проекты, которые должны быть включены в пакет. Разработанная система сборки также включает в архив информацию о версии и содержимом архива. При этом объектные файлы, заголовки, или другие временные файлы, необходимые для сборки, в программный пакет не включаются.

Нередко в программный пакет нужно включить результат сборки сразу нескольких конфигураций, например – отладочная конфигурация и конфигурация с оптимизацией исполняемых файлов. Разработанная система сборки позволяет указать, какие конфигурации необходимо собрать и включить в пакет.

Система сборки должна учитывать, что файлы продукта могут занимать очень много места на диске (распакованный архив может занимать несколько гигабайт места на диске), поэтому во время создания программного пакета копирование файлов во временный каталог нежелательно. Чтобы избежать копирования, разработанная система сборки еще на этапе сборки размещает файлы согласно иерархии каталогов, которая будут использоваться в пакете.

3.10 Создание пакета обновлений

Для обновления больших программных продуктов часто используются специальные пакеты обновлений. В работе был разработан способ автоматизировать создание пакета обновлений новой версии программного продукта. В данном случае пакет обновлений – архив, такой же, как и программный пакет, с той лишь разницей что в пакете обновлений содержатся не все файлы, а только те, которые необходимо заменить.

Преимущества использования пакета обновлений вместо полных программных пакетов для каждой новой версии:

1. Скачивание и установка пакета обновлений занимает меньше времени, чем скачивание и установка полного пакета.
2. В случае, когда необходимо одновременно иметь несколько версий продукта, использование пакетов обновлений приводит к значительной экономии места на диске.

Пакет обновлений между двумя ревизиями системы контроля версий должен содержать все соответствующие изменения. Если вручную указывать, какие файлы должны войти в пакет обновлений, то велика вероятность совершить ошибку и забыть указать некоторые файлы. С другой стороны, невозможно однозначно

определить до сборки, какие именно файлы будут созданы и изменены во время сборки, так как для этого нужно вручную указывать, какие файлы изменяются вызываемыми инструментами сборки. Поэтому информация о том, какие файлы исходного кода были изменены между ревизиями, недостаточна для того, чтобы получить список файлов, которые нужно включить в пакет обновлений.

В этой связи в разрабатываемой системе сборки был реализован механизм, позволяющий автоматизировать этот процесс следующим образом:

1. Собирается полный программный пакет для первой ревизии
2. Собирается полный программный пакет для второй ревизии
3. С помощью команды diff -qr сравнивается распакованное содержимое программных пакетов, и в пакет обновлений включаются только новые и измененные файлы. Реализация представлена в разделе Приложение 3.

Чаще всего пакеты обновлений создаются последовательно для новых ревизий, то есть вторая "конечная" ревизия одного пакета обновлений является первой "начальной" ревизией для следующего пакета обновлений. Соответственно, если при сборке пакета обновлений на диске остались результаты сборки предыдущего пакета обновлений, то собирать полный программный пакет для первой ревизии не обязательно. Кроме того, так как система сборки собирает только измененные проекты, то сборка второго программного пакета не занимает много времени. Таким образом создание пакета обновлений занимает меньше времени, чем сборка полного программного пакета.

ЗАКЛЮЧЕНИЕ

Целью работы являлась разработка системы сборки программного обеспечения, позволяющей ускорить сборку C/C++ проектов. Для достижения заданной цели в рамках работы были рассмотрены походы к сборке программного обеспечения, проанализированы существующие решения, и выдвинуты требования к разрабатываемой системе сборки.

Результатом работы стала система сборки, включающая функциональность, не реализованную в существующих альтернативных решениях:

- Реализован механизм кэширования результатов компиляции, позволяющий многократно ускорить повторную сборку при активной работе с несколькими ветками системы контроля версий.
- Реализован механизм распределённой компиляции, позволяющий задействовать дополнительные вычислительные мощности удаленных машин. При этом накладные расходы составляют около 10-12% от времени сборки.
- Предложен и реализован автоматизированный метод создания пакетов обновления для программных продуктов.

В рамках работы также был предложен и реализован метод интеграции предкомпилированных заголовков в процесс сборки, позволяющий ускорить сборку проектов до 2 раз, и реализован метод выполнения модульного тестирования во время сборки.

Исследования, проведенные в работе, опираются на случаи использования реальных проектов программного обеспечения промышленного масштаба. Разработанная система сборки применяется для сборки коммерческого продукта Tbricks by Itiviti.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Kumfert, G., Epperly, T., Software in the DOE: The Hidden Overhead of “The Build” / Lawrence Livermore National Lab. – 2002.
2. Smith, P., Software Build Systems: Principles and Experience / Смит П. – Addison-Wesley, 2011. – 621 с.
3. Hähne, L., Empirical Comparison of SCons and GNU Make / Technical University Dresden. – 2008.
4. Miller, P., Recursive Make Considered Harmful / AUUGN Journal of AUUG Inc. – 1998.
5. ccache – a fast C/C++ compiler cache [Электронный ресурс]. – Режим доступа: https://ccache.samba.org/. – (Дата обращения 01.05.2018).
6. Amdahl, Gene M., Validity of the single processor approach to achieving large-scale computing capabilities / Proceeding of AFIPS '67 (Spring). – 1967.
7. Distributed builds for C, C++ and Objective C [Электронный Ресурс]. – Режим доступа: https://github.com/distcc/distcc/. – (Дата обращения 02.05.2018).
8. GNU Make build tool [Электронный Ресурс]. – Режим доступа: https://www.gnu.org/software/make/. – (Дата обращения 03.05.2018).
9. Apache Ant build tool [Электронный ресурс]. – Режим доступа: https://ant.apache.org/. – (Дата обращения 04.05.2018).
10. Apache Maven build tool [Электронный ресурс]. – Режим доступа: https://maven.apache.org/. – (Дата обращения 05.05.2018).
11. CMake build tool [Электронный ресурс]. – Режим доступа: https://cmake.org/. – (Дата обращения 06.05.2018).
12. Gradle build tool [Электронный ресурс]. – Режим доступа: https://gradle.org/. – (Дата обращения 06.05.2018).
13. Bazel build tool [Электронный ресурс]. – Режим доступа: https://bazel.build/. – (Дата обращения 06.05.2018).

ПРИЛОЖЕНИЕ 1

Вывод результатов модульного тестирования во время сборки:

```
1 $ tbmake
2 --- [tbricks_util] Build Started : dynamic 64-bit
  library for linux
3 --- [tbricks_util] Build Finished : dynamic 64-bit
  library for linux
4 --- [util_test] Build Started : dynamic 64-bit program
  for linux
5 #   install  symlinks
6 #   ./../../../../build.x86_64-unknown-
  linux/bin/util_test -> server.sh
7 --- [util_test] Build Finished : dynamic 64-bit program
  for linux
8 --- [util_test] Execute
9 [==========] Running 118 tests from 25 test cases.
10  [----------] Global test environment set-up.
11  [----------] 3 tests from ArrayTest
12  [ RUN      ] ArrayTest.tuple_to_array
13  [       OK ] ArrayTest.tuple_to_array (0 ms)
14  [ RUN      ] ArrayTest.make_array
15  [       OK ] ArrayTest.make_array (0 ms)
16  [ RUN      ] ArrayTest.carray_to_array
17  [       OK ] ArrayTest.carray_to_array (0 ms)
18  [----------] 3 tests from ArrayTest (0 ms total)
19
20
21
22  [----------] Global test environment tear-down
```

```
23  [==========] 118 tests from 25 test cases ran. (126 ms
  total)
24  [  PASSED  ] 118 tests.
25  --- [util_test] Passed
```

ПРИЛОЖЕНИЕ 2

Содержимое предкомпилированного заголовочного файла pch.h:

```
1 #pragma once
2
3 #ifdef PCH_ENABLED
4
5 #include <algorithm>
6 #include <deque>
7 #include <iostream>
8 #include <list>
9 #include <map>
10  #include <memory>
11  #include <set>
12  #include <string>
13  #include <sstream>
14  #include <vector>
15
16  #include "strategy/Printable.h"
17
18  #include "protocol/cpp/fields.h"
19
20  #endif
```

ПРИЛОЖЕНИЕ 3

Команды сравнения двух программных пакетов:

```
1 .PHONY: diff
2 diff: build
3   @echo ------------------------------------------
4   @echo GENERATING PATCH DIFF
5   @echo ------------------------------------------
6   diff -qr --unidirectional-new-file \
7         $(PATCH_ROOT)/$(SOURCE_REV) \
8         $(PATCH_ROOT)/$(TARGET_REV) \
9         > $(PATCH_ROOT)/diff.raw \
10          ; rc=$$?; if [ $$rc -eq 1 ]; then exit 0; else
  exit $$rc; fi
11      cat $(PATCH_ROOT)/diff.raw \
12          | grep 'Files .* differ' \
13          | awk '{print $$2}' \
14          | sed 's@$(PATCH_ROOT)/$(SOURCE_REV)/@@' \
15          | grep -v -e 'DESCRIPTION' -e 'status/' -e
  'asan/status/' -e 'debug/status/' \
16          | tee $(PATCH_ROOT)/diff.cooked.paths \
17          | sed 's@.*/@@' \
18          > $(PATCH_ROOT)/diff.cooked.names
19      grep -rlf $(PATCH_ROOT)/diff.cooked.names \
20          $(PATCH_ROOT)/$(TARGET_REV)/status  \
21          | sed 's@.info/lname@@' \
22          | sed
  's@$(PATCH_ROOT)/$(TARGET_REV)/status/src/@@' \
23          > $(PATCH_ROOT)/diff.raw.projs
24      echo libraries/util \
```

```
25          >> $(PATCH_ROOT)/diff.raw.projs
26      cat $(PATCH_ROOT)/diff.raw.projs | sort | uniq \
27          > $(PATCH_ROOT)/diff.cooked.projs
```