

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технологический университет «СТАНКИН»**

**(ФГБОУ ВО «МГТУ «СТАНКИН»)**

|  |  |
| --- | --- |
| **Институт**  **информационных**  **систем и технологий** | **Кафедра**  **информационных систем** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Махмудов Бобурбек Нодирбекович** | | | |
| Тема: «**Исследование методов повышения производительности систем веб-шаблонов и разработка системы шаблонизаци на их основе**» | | | |
| **Выпускная квалификационная работа на присвоение квалификации «бакалавр» по направлению 09.03.02 «Информационные системы и технологии»** | | |
|  | Регистрационный № \_\_\_\_\_\_\_ | |
| **Заведующий кафедрой**  **д.т.н., проф.** |  | **Позднеев Б.М.** |
|  | подпись |  |
| **Руководитель**  **к.т.н., доцент** |  | **Бумарин Д.П.** |
|  | подпись |  |
| **Студент** |  | **Махмудов Б.Н.** |
|  | подпись |  |
|  |  |

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc34511621)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 4](#_Toc34511622)

[1.1. Термины и определения 4](#_Toc34511623)

[1.2. Общий принцип работы систем веб-шаблонов 7](#_Toc34511624)

[1.3. Обзор существующих решений 11](#_Toc34511625)

[1.4. Основные причины низкой производительности шаблонизаторов 15](#_Toc34511626)

[1.5. Цели и задачи 21](#_Toc34511627)

[ГЛАВА 2. ИССЛЕДОВАНИЕ ВОЗМОЖНЫХ СПОСОБОВ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ СИСТЕМ ВЕБ ШАБЛОНОВ 23](#_Toc34511628)

[2.1. Устранение причины низкой производительности, связанной с использованием интерпретаторов 23](#_Toc34511629)

[2.2. Уменьшение системных вызовов 25](#_Toc34511630)

[2.3. Решение проблемы простоя системных ресурсов 31](#_Toc34511631)

[2.4. RESTful сервис, выполняющий роль системы веб шаблонов 31](#_Toc34511632)

[ГЛАВА 3. ПРОЕКТИРОВАНИЕ СЕРВИСА ПО ПРЕДОСТАВЛЕНИЮ УСЛУГ СИСТЕМЫ ВЕБ-ШАБЛОНОВ 32](#_Toc34511633)

[3.1. Моделирование работы сервиса 32](#_Toc34511634)

[3.2. Выбор инструментов 32](#_Toc34511635)

[3.3. Разработка сервиса 32](#_Toc34511636)

[3.4. Результат разработки, оценка производительности 32](#_Toc34511637)

[3.5. Потенциал к масштабированию 38](#_Toc34511638)

[ЗАКЛЮЧЕНИЕ 39](#_Toc34511639)

[СПИСОК ЛИТЕРАТУРЫ 40](#_Toc34511640)

[ПРИЛОЖЕНИЕ А. 42](#_Toc34511641)

[ПРИЛОЖЕНИЕ Б. 43](#_Toc34511642)

ВВЕДЕНИЕ

# АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## Термины и определения

Предметной областью данной выпускной квалификационной работы является применение систем веб-шаблонов для массового создания веб-документов. В работе используется ряд терминов, определение которых приведено ниже.

Система – комбинация взаимодействующих элементов, организованных для достижения одной или нескольких поставленных целей [1].

Веб-шаблон – HTML-код с готовым дизайном и версткой, а также с дополнительной разметкой на языке шаблонизации, который используется для построения веб-документа.

Шаблонизатор – отдельное программное обеспечение или программный модуль, осуществляющий процесс генерации конечных веб-документов. Входными потоками процесса генерации являются веб-шаблон и контекст шаблонизации (данные), а роль управляющего потока выполняет формальная грамматика используемого языка шаблонизации.

Контекст шаблонизации – структура данных, которая содержит переменные окружения и методы, которые в свою очередь могут быть использованы шаблонизатором в процессе построения веб-документа.

Система веб-шаблонов (СВШ) – система, в состав которой входят три элемента:

1. веб-шаблон;
2. источника данных (JSON, XML, база данных);
3. шаблонизатор.

Шаблонизатор комбинирует данные и веб-шаблоны для массовой генерации веб-документов.

Лексический анализатор (англ. Lexical Analyzer) – отдельное программное обеспечение или программный модуль, предназначенный для аналитического разбора входной последовательности символов на распознанные группы –лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор», «оператор», «часть речи» и т. п.).

Синтаксический анализатор (англ. Parser) – отдельное программное обеспечение или программный модуль, предназначенный для сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализатором.

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, внутренние вершины которого поставлены в соответствие (помечены) с операторами языка программирования, а листья — с операндами [3].

Компиляция – сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоично-кодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы.

Интерпретация – процесс построчного анализа, обработки и выполнения исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения).

Среда выполнения (англ. execution environment) – вычислительное окружение, необходимое для выполнения компьютерной программы и доступное во время выполнения компьютерной программы. В среде выполнения, как правило, невозможно изменение исходного текста программы, но может наличествовать доступ к переменным окружения операционной системы, таблицам объектов и модулей разделяемых библиотек.

Сборка мусора (англ. garbage collection) в программировании – одна из форм автоматического управления памятью. Специальный процесс, называемый «сборщик мусора», периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложениями.

Системный вызов (англ. system call) в программировании и вычислительной технике – обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции, от имени этой программы.

Переключение контекста (англ. context switch) — в многозадачных ОС и средах - процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления или загрузки состояния задачи, к выполнению которой переходит процессор.

Асинхронный ввод/вывод – является формой неблокирующей обработки ввода/вывода, который позволяет процессу продолжить выполнение, не дожидаясь окончания передачи данных. Входные и выходные (I/O) операции на компьютере могут быть весьма медленными, по сравнению с обработкой данных. Устройство ввода/вывода может быть на несколько порядков медленнее, чем оперативная память. Например, во время дисковой операции, которой требуется десять миллисекунд для выполнения, процессор, который работает на частоте один гигагерц, может выполнить десять миллионов циклов команд обработки.

REST (от англ. Representational State Transfer – «передача состояния представления») – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful».

## Общий принцип работы систем веб-шаблонов

Все существующие системы веб-шаблонов функционируют похожим образом. Для работы системы необходимо выполнение следующих условий:

* наличие шаблонизатора – основной компонент системы, который генерирует конечный документ;
* наличием веб-шаблонов, размеченных языком шаблонизации, синтаксис которого поддерживается используемым шаблонизатором;
* наличие источника данных, при этом структура данных должна соответствовать правилам формирования контекста, который может быть обработан шаблонизатором.

При соблюдении вышеперечисленных условий, работу системы веб-шаблонов можно визуализировать на Рис. 1.2.1.



Рис. 1.2.1 Визуализация работы системы веб-шаблонов

Процесс шаблонизации в общем случае организован в последовательный набор операций, которые можно сгруппировать в четыре этапа:

1. Шаблон загружается в оперативную память. Источником шаблона может быть поток байт, файл, запись в базе данных и т.п.
2. Специальная подпрограмма, предназначенная для лексического анализа, именуемая «лексер», проводит анализ шаблона, и разбивает его на лексические единицы – лексемы, которые также называют токенами. Результатом этого процесса является поток токенов, которые значительно проще обрабатывать. Пример лексического разбора продемонстрирован на Рис. 1.2.1.
3. Другая подпрограмма, целью которой является синтаксический разбор потока токенов, именуемая «парсер», производит преобразование потока токенов в особую древовидную структуру данных, известной как абстрактное синтаксическое дерево.
4. На основе абстрактного синтаксического дерева и переданного контекста шаблонизации, шаблонизатор производит построение конечного веб-документа [3]. Пример построения абстрактного синтаксического дерева показан на Рис. 1.2.2.



Рис. 1.2.2 Лексический разбор шаблона, каждая лексема пронумерована



Рис. 1.2.3 Построение абстрактного синтаксического дерева

Далее можно ознакомится с функциональной моделью вышеописанного процесса в нотации IDEF0. На Рис. 1.2.4 представлена контекстная диаграмма процесса шаблонизации.



Рис. 1.2.4 Контекстная диаграмма процесса шаблонизации

На Рис. 1.2.5 приведена декомпозиция верхнего уровня процесса.



Рис. 1.2.5 Декомпозиция верхнего уровня процесса шаблонизации

## Обзор существующих решений

На сегодняшний день существует большое множество систем веб-шаблонов, как проприетарных, так и с открытым исходным кодом.

Системы веб шаблонов можно выделить в две основные категории, в зависимости от способа их размещения относительно конечного пользователя:

* server-side – работающие на стороне сервера;
* client-side – работающие в составе клиентского приложения.

Первые системы веб шаблонов работали на стороне сервера, и представляли из себя препроцессоры текста. Распространённой практикой было использование технологии CGI (Common Gateway Interface), которая подразумевает вызов веб-сервером внешнего программного кода, как правило скрипта, задачей которого была генерация веб-документа с использование данных, полученных в запросе от клиентского приложения. Со временем появились полноценные веб-фреймворки, которые имеют в своём составе систему веб-шаблонов в качестве отдельного программного модуля. Далее приведен перечень, наиболее популярных систем веб-шаблонов, работающих на стороне сервера.

Blade – входит в состав веб-фреймворка Laravel. Написан на языке PHP. Также, как и сам фреймворк, частью которого является, Blade обладает свободной лицензией и открытым исходным кодом.

Django – по сути сам является веб-фреймворком, написанном на языке Python, обладает широкими возможностями шаблонизации. На основе синтаксиса языка шаблонизации, используемого в Django было создано большое количество других СВШ. Свободная лицензия.

Mustache – это скорее спецификация по созданию систем веб-шаблонов, нежели отдельно взятый программный продукт, но существует не мало имплементаций данной спецификации на разных языках программирования. Синтаксис, определенный в спецификации, является одним из самых распространённых и широко-используемых. Открытая спецификация [4].

Twig – система веб-шаблонов, синтаксис которой основан на Django. Написан на языке PHP, наиболее часто используется при проектировании веб-приложений по паттерну MVC (Model View Controller). Свободная лицензия.

Lasso – является сервером приложений, имеющий в своем составе одноимённую подсистему веб-шаблонов. Является примером проприетарной СВШ. По возможностям значительно уступает свободным аналогам.

Также существуют системы веб-шаблонов, которые могут исполнятся прямо в клиентском приложении, как правило это веб-браузер. Основной принцип работы этой категории СВШ состоит в том, что они взаимодействуют с объектной моделью документа [5] (DOM), с целью изменения его внешнего представления. Достигается это путём использования сценарных языков, наиболее популярным из которых является JavaScript. Ниже приведён список систем веб-шаблонов, исполняемых на стороне клиента.

Handlebars – синтаксис этой системы веб-шаблонов вдохновлён спецификацией Mustache, и имеет с ней полную обратную совместимость. Для ускорения генерации документа, Handlebars заблаговременно компилирует веб-шаблон в JavaScript код, что положительно сказывается на его производительности [6].

Squirrelly – легковесная встраиваемая система веб-шаблонов, написанная как библиотека на языке JavaScript, обладает высокими показателями быстродействия, эффективности. Не смотря на свой маленький размер, Squirrelly предоставляет широкий спектр возможностей шаблонизации.

Dot.js – по заявлению разработчиков является самым быстрым и лаконичным из всех существующих шаблонизаторов, написанных на JavaScript. Но из-за своей легковесности, не предоставляет всех функциональных возможностей аналогов.

Все приведённые выше системы веб-шаблонов были разработаны для решения определённых задач, каждый из которых обладает рядом достоинств и недостатков. Выбор той или иной системы веб-шаблонов, да и любого программного средства в целом, как правило, обусловлен несколькими факторами, среди которых можно выделить:

* Уровень навыков и опыт команды разработчиков в применении определённой технологии. Разработчики часто отдают свои предпочтения уже известным и проверенным решениям, так как внедрение новой технологии всегда подразумевает определённые риски, связанные с неопределённостью.
* Окружение. А именно какой перечень технологий уже используется, и как хорошо новое программное средство будет взаимодействовать с существующим окружением.
* Удобство в использовании. Насколько продукт понимаемый, изучаемый, используемый и привлекательный для [пользователя](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C) в заданных условиях [2]. Иначе говоря, пользователь должен иметь возможность эксплуатировать программное средство в определенных условиях для достижения установленных целей с необходимой результативностью, эффективностью и удовлетворённостью.
* Удобство в сопровождении. Сколько усилий нужно прилагать для доработок и поддержки программного продукта в соответствии с изменяющимися требования заказчиков.
* Производительность. Насколько быстро работает программное обеспечение под определённой нагрузкой. Часто пренебрегаемый критерий выбора, так как на практике его перевешивают предыдущие пункты. Это связано с тем что быстродействие системы веб-шаблонов не является «узким местом» для большинства надсистем в которых она функционирует, и время, затрачиваемое на выполнение ею полезной работы, находится в «приемлемых» пределах.
* Эффективность. Насколько достигнутые результаты соотносятся с затраченными ресурсами. Немаловажный фактор, в связи с тем, что параллельно с системой веб-шаблонов могут исполнятся другие программные средства, которые также потребляют системные ресурсы.

Именно производительность и эффективность, и являются центральной темой исследования данной выпускной квалификационной работы.

## Основные причины низкой производительности шаблонизаторов

В разделе «Обзор существующих решений» был приведён перечень самых широко-используемых систем веб шаблонов, проанализировав эти решения, можно сделать вывод о том, что большинство из них имеет ряд общих признаков, тем или иным образом влияющие на производительность системы в целом:

1. они имеют в своем составе шаблонизатор, написанный на интерпретируемом языке;
2. имплементация шаблонизатора, часто использует системные вызовы;
3. наличие сборщика мусора, входящее в состав среды исполнения;
4. необходимость заново проводить шаги загрузки, лексического и синтаксического анализов при построение нового шаблона;
5. простой ресурсов многоядерных процессоров.

Написание шаблонизаторов на интерпретируемых языках, пожалуй, является самым значительным из факторов, вносящих вклад в снижение производительности и эффективности функционирования систем веб-шаблонов. Программный код, написанный на интерпретируемых языках значительно уступает компилируемым аналогам в быстродействии, в связи с тем, что в ходе выполнения процесса интерпретации, присутствует дополнительный шаг. Дополнительный шаг представляет из себя необходимость построчной трансляции написанного кода в промежуточное представление, и его последующей интерпретации в машинный код. Из этого следует потребность в достаточно сложной и большой среде выполнения, что требует дополнительных пространственных и временных ресурсов вычислительной системы. Также тот факт, что трансляция происходит построчно, прямо во время исполнения программного кода, делает невозможным применение достаточно большого количества оптимизаций, который могут производить компиляторы, связанные с уменьшением времени выполнения программы, размера исполняемого файла, или энергопотребления в ходе её исполнения. Наглядно разницу между процессами компиляции и интерпретации можно увидеть на Рис. 1.4.1. Достоинством таких языков, является наличие различного рода высокоуровневых абстракций, облегчающих процесс кодирования, а также простота в отладке и поддержке программного кода. Они идеально подходят для разработок по модели быстрого прототипирования.



Рис. 1.4.1 Сравнение процессов компиляции и интерпретации

Частое выделение памяти из управляемой кучи, и копирование данных из одной области памяти в другую, является ещё одним немаловажным фактором оказывающее существенное влияние на снижение быстродействия. Это часто требуемая операция при работе со строками, например, их соединения. Также если памяти в управляемой куче недостаточно, то будет использован системный вызов, для расширения кучи. Использование системного вызова, всегда подразумевает смену контекста исполнения, то есть управление процессором переходит операционной системе. Смена контекста, сам по себе является «дорогой» операцией, так как представляет из себя сохранение состояния исполняемой программы, с последующим восстановлением этого состояния. Во время процесса сохранения и загрузки состояния нитей исполнения, не выполняется никакой полезной работы, соответственно это оказывает негативное влияние на общее время выполнения программы. Также использование системных вызовов, является не детерминированными во времени, то есть исполнение одного и того же системного вызова, вообще говоря, может занимать разное количество времени. С графической схемой смены контекста можно ознакомится на Рис. 1.4.2.



Рис. 1.4.2 Графическая схема смены контекста

Сборщик мусора является неотъемлемой частью многих языков программирования, как интерпретируемых, так и компилируемых. Его наличие значительно снижает, а иногда и вовсе отменяет необходимость в ручном управлении памятью, а именно её выделение и последующее освобождение. Он входит в состав среды выполнения программного обеспечения, и хранит сведения обо всей памяти которая в данный момент используется в программе, делается это с помощью подсчёта ссылок на конкретную область или «объект». Регулярно сборщик мусора сканирует управляемую кучу на наличие «осиротевших» областей памяти, и освобождает их. Ещё одной задачей сборщика мусора может являться дефрагментация управляемой кучи, при невозможности выделения из неё памяти. Все эти процессы происходят в фоновом режиме, и не требуют вмешательств со стороны разработчика. Но как сканирование, так и дефрагментация в особенности, являются очень затратными по отношению к системным ресурсам операциями, в следствии чего производительность шаблонизатора существенно деградирует. Принцип работы, основанный на подсчёте активных ссылок на объекты в управляемой куче проиллюстрирован на Рис. 1.4.3.

Большинство систем веб-шаблонов не имеют или не используют по умолчанию механизм кеширования уже скомпилированного веб-шаблона, это в свою очередь может привести к ситуациям, где все три шага шаблонизации, предшествующие шагу построения документа приходится проводить каждый раз, когда меняется текущий шаблон. Если система веб-шаблонов используется в среде, где смена шаблона происходит с высокой частотой, то необходимость проводить загрузку, лексический и синтаксические анализы шаблона большое количество итераций, может привести к критически низким показателям производительности.



Рис. 1.4.3 Принцип работы «сборщика мусора»

Простой ресурсов многоядерных процессоров – это проблема, возникающая ввиду сильной связанности шагов процесса шаблонизации и невозможностью разделения данной задачи на подзадачи между несколькими нитями исполнения. В вычислительных системах с несколькими центральными процессорами, и отсутствием иного рода вычислительных задач, это может привести к простою или бездействию остальных процессоров, при этом время работы самого шаблонизатора может увеличиться почти пропорционально количеству бездействующих вычислительных единиц.

Приняв во внимание вышеперечисленные проблемы и их специфику можно сформировать перечень целей и задач, которые необходимо достичь и решить соответственно, в рамках данной выпускной квалификационной работы.

## Цели и задачи

Как говорилось ранее, в данной работе основное внимание отводится производительности и эффективности систем веб-шаблонов. В разделе «Основные причины низкой производительности шаблонизаторов» был приведён и подробно разобран перечень факторов, оказывающих негативное влияние на быстродействие системы веб-шаблонов в целом. Отталкиваясь от этой информации, можно сформулировать основные задачи, который предстоит решить в рамках этой выпускной квалификационной работы:

1. Исследовать альтернативные способы написания шаблонизаторов на компилируемых языках программирования, при этом необходимо минимизировать отрицательное воздействие такого подхода на такие показатели качества, как удобство в использовании и простота сопровождения программного продукта.
2. Изучить парадигмы и практики в программировании, которые позволяют уменьшить количество операций, связанных с выделением памяти и копированием данных между разными её областями. При этом не должно происходить значительного усложнения модели управления памятью вычислительной системы.
3. Проанализировать существующие языки программирования, которые не реализуют управление памятью через «сборку мусора». Оценить предоставляемые ими возможности, и сделать вывод в целесообразности отказа от сборщика мусора.
4. Исследовать методологии и концепции кеширования результатов уже осуществлённых вычислений, с целью минимизации накладных расходов, связанных с необходимостью повторного выполнения всех шагов шаблонизации, возникающих как следствие смены текущего веб-шаблона.
5. Ознакомится с парадигмой многопоточного программирования и изучить сложности, которые могут возникать при написании многопоточных программ. Также необходимо исследовать концепцию асинхронного программирования, оценить возможность её применения для ещё более эффективного использования системных ресурсов.

Описанные выше задачи, отражают пять основных проблем низкой производительности шаблонизаторов. В ходе решения этих задач будет собрана большая база знаний, которые в дальнейшем могут быть использованы для реализации высокопроизводительной, и в то же время удобной в использовании системы веб-шаблонов, что в свою очередь является главной целью данной выпускной квалификационной работы.

# ИССЛЕДОВАНИЕ ВОЗМОЖНЫХ СПОСОБОВ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ СИСТЕМ ВЕБ ШАБЛОНОВ

## Устранение причины низкой производительности, связанной с использованием интерпретаторов

Так как использование любых языков программирования, нуждающиеся в предварительной интерпретации для выполнения кода не допустимо для достижения повышенной производительности, то нужно искать альтернативы среди компилируемых языков программирования.

Компилируемый язык программирования – язык программирования, исходный код которого преобразуется компилятором в машинный код и записывается в файл с особым заголовком и/или расширением для последующей идентификации этого файла, как исполняемого операционной системой, в отличие от интерпретируемых языков программирования, чьи программы выполняются пошагово программой-интерпретатором.

Языки программирования принято разделять на компилируемые и интерпретируемые в силу типичных различий:

* скорость выполнения программы, скомпилированной в машинный код, превосходит скорость интерпретируемой программы, как правило, в десятки и сотни раз;
* в случае использования компилятора, при внесении изменений в исходный код программы, прежде чем эти изменения можно будет увидеть в работе программы, необходимо выполнить повторную компиляцию исходного текста.

Также среди компилируемых языков принято выделять две различных категории в зависимости от выходных данных компиляции:

1. языки, которые компилируются в машинный код целевой архитектуры вычислительной машины;
2. языки, которые компилируются в машинный код некой виртуальной машины, также известный как байт-код.

Программы, написанные на языках, компилируемые в машинный код целевой вычислительной машины очень компактны, не требуют вспомогательных средств окружения для исполнения, не зависят от исходного кода после процесса компиляции и самое главное, они обладают высокими показателями быстродействия. Их основной недостаток – это низкий уровень переносимости, из-за необходимости перекомпилировать, а возможно и адаптировать исходный код под каждую целевую архитектуру вычислительных машин. Примерами таких языков являются:

* C,
* C++,
* Haskell,
* Go,
* Rust,
* Swift.

Что же касается языков, компилируемые в байт-код виртуальной машины, то для исполнения они требуют эту самую виртуальную машину, могут выполнятся на любой архитектуре, где есть виртуальная машина. По быстродействию в разы уступают аналогам, компилируемых под конкретную архитектуру. Основной недостаток – это необходимость наличия виртуальной машины, которая также негативно влияет на производительность исполняемой программы. Примерами таких языков являются:

* Java,
* C#,
* Erlang.

Приняв во внимание вышеописанные различия среди компилируемых языков, а также тот факт, что основная задача – это повышение производительности, то выбор языка будет осуществляться в категории компилируемых в машинный код целевой архитектуры.

В целях облегчения разработки, и упрощения сопровождения проектируемой системы веб-шаблонов, выбираемый язык должен удовлетворять следующим требованиям:

* наличие зрелой экосистемы вокруг языка, то есть активное сообщество, наличие библиотек для решения часто возникающих задач;
* наличие какого-либо программного механизма для управления пакетами и зависимостями между ними;
* поддержка обобщённого программирования и какой-либо формы полиморфизма;
* наличие встроенных механизмов документирования кода.

В следующих разделах будут исследованы другие способы повышения производительности системы веб-шаблонов, и как следствие будут предъявлены дополнительные требования к другим аспектам выбираемого языка программирования.

## Уменьшение операций по выделению памяти

В высокоуровневых языках программирования, частое выделение памяти из управляемой кучи, связано с абстракциями над ссылочными типами данных, которые призваны упростить процесс программирования, за счёт сокрытия от пользователя нижележащих механизмов управления памятью. Но ценой таких абстракций является лишение того уровня контроля над памятью, которое возможно в более низкоуровневых языках, в которых ссылочные типы данных доступны. Преимущество ссылочных типов в том, что программист сам решает, когда ему нужно создать новый экземпляр некоторого объекта, выделив для этого память, и когда можно просто сослаться на уже существующий объект, посредством его адреса в памяти. Недостаток такого подхода заключается в увеличении количества ошибок в коде, которые часто бывает сложно отлаживать, такие как «утечки» памяти или двойные освобождения. Не смотря на подверженность ошибкам, использование ссылочных типов данных, вместо создания нового экземпляра объекта, при работе с большим количеством строковых данных, может положительно сказаться на быстродействии.

Ещё одним способом снижения количества выделений памяти, является использование предварительного выделения достаточного для целевой задачи объема памяти, так как время, затраченное на выделение, не зависит от объема запрашиваемой памяти. В результате можно сэкономить на количествах запросах на выделение памяти. Это в свою очередь, подразумевает тщательный анализ вычислительной задачи, с целью определения нижних и верхних пределов объема памяти, требуемой на её выполнение, например, на основе статистических данных.

На основе этих наблюдений, можно сформулировать дополнительные требования к выбираемому языку программирования:

* наличие поддержки ссылочных типов данных;
* наличие встроенных механизмов, снижающие вероятность возникновения ошибок, связанных со ссылочными типами данных.

Среди компилируемых языков, которые предоставляют ссылочные типы данных и соответственно более высокий уровень контроля над памятью, доступной исполняемой программе, можно выделить: С/С++, Rust, Go, Haskell.

C и C++ являются пожалуй самыми старыми из перечисленных языков, их отличительной особенностью является принцип «trust the programmer», т.е. «доверяй программисту», это подразумевает то, что язык даёт полный контроль над вычислительными ресурсами машины. Потенциально это позволяет писать очень быстрый код, с маленьким размером исполняемого файла. Но как оказывается на практике, ввиду отсутствия в данных языках каких-либо механизмов безопасности, написанные программы подвержены появлению различного рода, сложно-отслеживаемых и отлаживаемых ошибок.

Haskell – стандартизированный чистый функциональный язык программирования общего назначения. Является одним из самых распространённых языков программирования с поддержкой отложенных вычислений. В силу того, что язык поддерживает только функциональную парадигму программирования, написание на нём программ, является не лёгкой задачей для тех, кто пришёл из более традиционного процедурного или объектно-ориентированного языка программирования. Ещё одной особенностью Haskell является то, что все типы данных неизменяемые, что подразумевает создание нового объекта в памяти каждый раз, когда нужно мутировать данные.

Go – компилируемый многопоточный язык программирования, разработанный внутри компании Google. Разрабатывался как язык программирования для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах. Он может рассматриваться как попытка создать замену языкам Си и C++ с учётом изменившихся компьютерных технологий и накопленного опыта разработки крупных систем. Является хорошим кандидатом для написания высокопроизводительных систем. Единственным недостатком, связанным с целями данной работы, является наличие сборщика мусора в среде выполнения, который будет подробно разобран в следующем разделе.

Rust – мультипарадигмальный компилируемый язык программирования общего назначения, сочетающий парадигмы функционального и процедурного программирования с объектной системой, основанной на типажах, и с управлением памятью через понятие «владения», что позволяет обходится без сборки мусора. Язык с самого начала проектировался для использования в системном программировании, где производительная эффективность является одной из самых важных показателей качества. В отличие от языков С/С++, безопасность в смысле снижения количества ошибок, заложена в сам язык на фундаментальном уровне. Язык имеет относительно высокий порог обучения.

Далее будет разобран механизм сборки мусора, и возможные решения, которые позволяют обходится без него, с минимальными потерями в плане продуктивности и удобства в использовании.

## Альтернатива сборщику мусора

Как было сказано ранее, сборка мусора, является неотъемлемой частью сред выполнения многих языков программирования. Но за удобство и простоту использования, которую предоставляет сборщик мусора, нередко приходится платить производительностью и высокими уровнями потребления памяти вычислительной системы. Это связано с тем, что для правильной работы, сборщик мусора ставит исполнение программы на «паузу», после чего выполняет освобождение памяти от неиспользуемых объектов, хотя длительность таких остановок невелика и современные сборщики мусора реализуют, оптимизированные на скорость выполнения, алгоритмы сканирования памяти на неиспользуемые объекты, тем не менее это не может не оказывать негативного воздействия на быстродействие всей программы в целом.

Существует несколько альтернативных методик программирования, которые позволяют обходится без сборки мусора вовсе, далее приведены наиболее часто применяемые из них.

Ручное управление памятью – самый простой и прямой способ, программист сам решает, когда ему нужно выделение памяти, а после её получения берёт на себя ответственность за её своевременное освобождение. Идеально подходит для небольших и несложных программ. Но с ростом числа строк кода, и сложности программной логики, такой подход к управлению памятью, ведёт к росту числа ошибок, некоторые из которых может быть достаточно сложно отладить.

Создание пула объектов – подход который использует методику предварительного выделения памяти для различных типов данных, используемых в ходе исполнения программы. Объекты создаются заранее в этой области памяти, и предназначены для многократного использования, что позволяет обходится без дальнейших выделений памяти. Достаточно сложный в реализации, так как при изменении типов данных программы, необходимо эти изменения учитывать и в механизме пула объектов.

Использование стека вместо управляемой кучи – суть подхода заключается в том, что если размерность всех типов данных известна заранее, то можно обойтись без выделения памяти вовсе, и создавать все объекты прямо на стеке, что является гораздо более быстрой операцией, и освобождает программиста от обязанности освобождать память, так как после завершения функции весь стек будет автоматически освобожден. Данный подход имеет крайне узкое применение, ввиду того, что на практике очень часто приходится работать с типами данных, размерность которых определяется во время выполнения программы. Но его можно использовать в связке с другими, в силу описанных выше преимуществ.

Получение ресурса есть инициализация (англ. Resource Acquisition Is Initialization (RAII)) – программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение – с уничтожением объекта. В первые идиома была реализована на языке С++, она требует от программиста тщательного проектирования классов объектов, так чтобы при окончании времени жизни переменной, «владеющей» экземпляром класса, этот экземпляр класса также был уничтожен. Идиома сложна в реализации, но при правильном использовании, сильно упрощает написание кода, без отрицательных эффектов на произвольность программы.

Проанализировав альтернативные подходы к управлению памятью, можно дополнить перечень требований к выбираемому языку программирования:

* отсутствие механизма сборки мусора, или наличие возможности её отключения;
* наличие программных механизмов, реализации идиомы RAII для управления памятью.

В данном разделе были изучены существующие способы управления памятью без сборщика мусора. Можно сделать вывод о том, написание программ без использования сборки мусора, не только возможно, но и может положительно сказаться на производительности и эффективности, разрабатываемого программного обеспечения.

## Кеширование наиболее часто используемых шаблонов

Необходимость в кешировании часто возникает в самых различных задачах, связанных с разработкой программного обеспечения. Основная идея кеширования заключается в сохранении результата какой-либо вычислительной операции для дальнейшего его использования из локального хранилища «кеша». При этом происходит экономия времени за счёт отсутствия потребности в повторном выполнении этой вычислительной операции, но возрастает потреблении пространственных ресурсов. Для решения проблемы повторного выполнения всех шагов шаблонизации, можно использовать локальный для шаблонизатора «кеш», который будет содержать уже построенные абстрактные синтаксические деревья, предыдущих шаблонов. В этом случае, если поступит запрос на генерацию веб-документа на основе веб-шаблона, который ранее был уже использован, то все шаги шаблонизации предшествующие построению веб-документа, можно пропустить, так как абстрактное синтаксическое дерево уже имеется в кеше.

В результате использования кеша, возникают две проблемы:

1. необходимость поддерживать когерентность кеша;
2. необходимость ограничить размер кеша.

Суть проблемы когерентности кеша, заключается в том, что данные в кеше должны соответствовать тем данным, на основе которых они были получены. Решением данной проблемы, в случае системы веб-шаблонов, является триггерное обновление кеша при изменении шаблона, на основе которого было построено абстрактное синтаксическое дерево. Обновление кеша необходимо проводить с блокировкой шаблонизатора, с целью исключить вероятность использования кеша другими нитями исполнения.

Необходимость ограничить размер кеша, возникает как результат фиксированности размеров памяти, доступной для использования шаблонизатором, и потенциально неограниченным количеством шаблонов, имеющимся в системе. Для решения данной проблемы можно использовать алгоритм вытесняющего кеширования LRU (Last Recently Used – Вытеснение давно неиспользуемых). Суть алгоритма заключается в том, что заранее задается размер кеша, в случае шаблонизатора это максимальное количество АСД, которые могут находится в кеше. С каждым вхождением в кеше, ассоциируется «бит возраста», который при обращении к этому вхождению становится равным нулю, а для всех остальных вхождений увеличивается на единицу. В результате, при переполнении кеша, и необходимости загрузить в него новое АСД, то вхождение с наибольшим «возрастом» вытесняется из кеша.

## Решение проблемы простоя системных ресурсов

В следующем разделе будет выбран язык программирования для написания системы веб-шаблонов, с учётом всех вышеприведённых требований.

## Выбор языка программирования

## RESTful сервис, выполняющий роль системы веб шаблонов

# ПРОЕКТИРОВАНИЕ СЕРВИСА ПО ПРЕДОСТАВЛЕНИЮ УСЛУГ СИСТЕМЫ ВЕБ-ШАБЛОНОВ

## Моделирование работы сервиса

## Выбор инструментов

## Разработка сервиса

## Результат разработки, оценка производительности



Рис. 3.4.1 Тест №1, 50 одновременных подключений



Рис. 3.4.2 Тест №1, настройки нагрузки



Рис. 3.4.3 Состояние системы, тест № 1, начало



Рис. 3.4.4 Состояние системы, тест № 1, пиковая нагрузка



Рис. 3.4.5 Тест №2, 100 одновременных подключений



Рис. 3.4.6 Тест №2, состояние системы, пиковая нагрузка



Рис. 3.4.7 Тест № 3, 200 одновременных подключений



Рис. 3.4.8 Тест №3, состояние системы, пиковая нагрузка



Рис. 3.4.9 Тест № 4, 550 одновременных подключений



Рис. 3.4.10 Тест № 4, состояние системы, одно ядро полностью нагружено



Рис. 3.4.11 Тест № 5, 700 подключений, деградация производительности



Рис. 3.4.12 Тест № 5, наличие ошибок в ответе от сервиса



Рис. 3.4.13 Тест №5, состояние системы, оба ядра полностью нагружены

## Потенциал к масштабированию

ЗАКЛЮЧЕНИЕ

СПИСОК ЛИТЕРАТУРЫ

1. ГОСТ Р ИСО/МЭК 15288-2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем.
2. ГОСТ Р ИСО/МЭК 25010-2015 Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.
3. Михаил Вюрш. Улучшение распознавания изменений в исходных кодах с помощью абстрактных синтаксических деревьев: дипломная работа [Текст]. Университет Цюриха 2006. – 64 с.
4. Twig Internals – Documentation [электронный ресурс]. Режим доступа: <https://twig.symfony.com/doc/2.x/internals.html>, свободный (дата обращения: 07.03.2020).
5. Chris Wanstrath. Mustache – Logic-less templates [электронный ресурс]: 2009. Режим доступа: <https://mustache.github.io/mustache.5.html>, свободный (дата обращения: 07.03.2020).
6. Flanagan, David. JavaScript: The Definitive Guide [Текст]. O'Reilly & Associates, 2006. – 992 с.
7. Introduction | Handlebars [электронный ресурс]. Режим доступа: <https://handlebarsjs.com/guide/#what-is-handlebars>, свободный (дата обращения: 07.03.2020).
8. How does memory allocation work in Python [электронный ресурс]. Режим доступа: <https://medium.com/datadriveninvestor/how-does-memory-allocation-work-in-python-and-other-languages-d2d8a9398543>, свободный (дата обращения: 14.03.2020).
9. Haskell and Rust [электронный ресурс]. Режим доступа: <https://www.fpcomplete.com/blog/2018/11/haskell-and-rust> (дата обращения 14.03.2020).
10. Блэнди Джим, Орендорф Джейсон. Программирование на языке Rust. – ДМК Пресс, 2018. – 550 с.
11. Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance – Computer, 1994. – 46 с.

ПРИЛОЖЕНИЕ А.

**ДОКУМЕНТАЦИЯ API**

ПРИЛОЖЕНИЕ Б.

**СИНТАКСИС ДЛЯ НАПИСАНИЯ ШАБЛОНОВ**