Министерство образования и науки Российской федерации

Федеральное государственное высшее образовательное учреждение
высшего профессионального образования
"Московский физико-технический институт
(государственный университет)"

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

Масштабирование асинхронных шаблонов программирования в больших проектах

Выпускная квалификационная работа
(бакалаврская работа)
Направление:03.03.01 Прикладные математика и физика

Выполнил:
студент 276 группы

Керимов Василий Алексеевич

Научный руководитель:

Соболев Артемий Анатольевич

Содержание

1. Постановка задачи 3
    1.1 Масштабируемость 3
    1.2 Асинхронное программирование 4
2. Обзор литературы 6
    2.1 Масштабируемость 6
    2.2 Инструментация 13
    2.3 NSOperation 15
3. Отчёт о проделанной работе 17
    3.1 Задача программы 17
    3.2 Детали реализации 17
    3.3 Инструментация 18
4. Заключение 19
5. Список литературы 20

Постановка задачи

Масштабируемость

Масштабируемость - важное свойство компьютерных систем, сетей и процессов. Оно предполагает простоту расширения набора аппаратных и программных средств для обеспечения возрастающих требований к системе. Более конкретно, под масштабируемостью подразумевают способность системы существенно увеличивать производительность при добавлении к ней новых вычислительных ресурсов(мощностей). Также под ним иногда подразумевают отсутствие необходимости в структурных изменениях кода программы для приспособления её к работе с добавленными ресурсами. Но там, где не оговорено иначе, мы всегда будем понимать под масштабируемостью её основное значение.

Необходимость в увеличении мощности какой-то системы, чтобы справится с растущим спросом на её использование, возникает достаточно часто. Также часто появляется необходимость в добавлении вычислительных ресурсов к системе для того, чтобы решить какую-то математическую задачу более точно. Эти примеры показывают, что масштабируемость важна как в промышленном, так и в научном программировании.

Для описания масштабируемости было придумано множество моделей и подходов, таких как, например, закон Амдаля и закон Густафсона, которые позволяют на основе имеющихся данных о поведении программы на системах с определённым количеством вычислительных ресурсов - например, процессоров или узлов сети, с достаточной точностью предсказать, как изменится быстродействие программы, если количество этих ресурсов изменится. Как правило, имеет смысл проводить эксперимент на машине с маленьким количеством процессоров, чтобы предсказать ускорение программы при увеличении их числа. По некоторым моделям можно сделать вывод о том, какого максимального ускорения можно будет в принципе достигнуть и при каком числе процессоров это произойдёт - неоценимая информация при решении о закупке новых мощностей для проекта.

Асинхронное программирование

Асинхронное программирование - подход к программированию, в котором нет блокирующих операции. Функции не ждут ответа от устройств ввода-вывода или веб-серверов, блокируя поток и не давая исполнятся другим функциям. Вместо этого ответ от вышеперечисленных устройств обрабатывается сразу по его получении специальными функциями обратного вызова. Таким образом, процесс никогда не блокируется, выполняя какую-то полезную работу, если она есть. Этот подход удобен случае, когда запросы к другим устройствам производятся очень часто - в некоторых случаях асинхронная программа в одном потоке может оказываться даже более эффективным, чем многопоточная, выполняющаяся с блокирующими операциями.

Асинхронный подход не обязан быть ограниченным одной нитью. Например, в набирающем популярность языке Swift подкласс Operation класса NSOperation, представленный на конференции разработчиков Apple WWDC в 2015 году в лекции "Advanced NSOperation" предлагает многопоточную реализацию этого шаблона.

В ней операции распределяются по очередям и начинают исполнятся по очереди, но при этом каждая отдельная операция может начать исполняться только после того, как выполнятся все её условия(такие как доступность сети, завершённость другой операции, от которой данная операция зависит, возможность выполнения только одной операции определённого типа одновременно и тому подобное). Таким образом, при эффективном дроблении задания программы на минимальные блоки, каждый из которых может исполняться автономно, не ждя ответа от устройств ввода-вывода, мы получаем как раз асинхронную программу. Поскольку асинхронные программы, как правило, имеют хорошую масштабируемость, можно предположить, что программы сделанные с использованием класса Operation обладают таким же свойством. Это может быть в особенности полезно не только для клиентских приложений, но и для, скажем, серверов, создание которых на языке Swift становится возможным благодаря таким разработкам как Swift Perfect.

Интересно проверить, насколько масштабируемость программы будет удовлетворять линейному закону, и оценить максимальную загруженность, которую сможет выдержать программа. Для этого была написана программа, которая делает измерения работы программы использующей Operation на машинах с процессорами разным количеством ядер и сравнивает, как полученные измерения соотносятся с существующими моделями масштабируемости, какая из них описывает их наиболее удачно.

Также, чтобы улучшить масштабируемость таких программ во втором смысле(отсутствие необходимости в структурных изменениях кода программы для приспособления её к работе с добавленными ресурсами) предлагается провести с помощью clang простейший статический анализ кода и указать на использование в проекте нежелательных шаблонов программирования(анти-паттернов).

Обзор литературы

Масштабируемость

Масштабируемость - общирное понятия, включающее в себя несколько разных концепций, разной сложности формализации и измерения.

Масштабирование делят на вертикальное и гризонтальное. Под вертикальным подразумевают увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий. Это самый простой способ масштабирования, так как не требует никаких изменений в прикладных программах, работающих на таких системах. Под горизонтальной же имеют в виду разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам (или их группам), и (или) увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом контексте означает возможность добавлять к системе новые узлы, серверы, процессоры для увеличения общей производительности. Этот способ масштабирования может требовать внесения изменений в программы, чтобы программы могли в полной мере пользоваться возросшим количеством ресурсов.

Кроме того, разделяют сильную и слабую масштабируемость: сильная масштабируемость ассматривает, как меняется время решения задачи с увеличением количества процессоров (или вычислительных узлов) при неизменном общем объёме задачи, а слабая - как меняется время решения задачи с увеличением количества процессоров (узлов) при неизменном объёме задачи для одного процессора (или узла).

Для измерения масштабируемости придумано несколько метрик, каждая из которых, в принципе, разумна, и предпочтение той или и ной альтернативам, как правило, объясняется особенносями того или иного проекта,

масштабируемостью которого интересуется, а также наличием или отсутствием о нём данных, нужных для рассчёта метрики.

Первой метрикой является ускорение - показатель, насколько сократилось время выполнения определённого объёма работы. То есть, для её измерения фиксируется определённый участок кода и измеряется его выполнения на машинах с разными мощностями.

$$S(p) = \frac{T(1)}{T(p)}$$

Эта одна из самых популярных метрик.

Следующей является увеличение масштаба - показатель, насколько увеличился объём работы, выполняемый за фиксированное время. В этом случае, соответственно, измеряется максимальное количество операций, которое может быть выполненно за фиксированный отрезок времени на разных машинах.

$$C(p) = \frac{X_{max}(p)}{X_{max}(1)}$$

Эта метрика часто используется при измерении масштабируемости веб-приложений, ведь основным требованием к ним как раз и является способность обслуживать возрастающее количество пользователей за то же время.

На самом деле, оба эти показателя означают одно и то же, потому что во сколько раз больше мы ускорили выполнение кода, в столько раз больше мы выполним его за одну единицу времени. Более формально, максимальное количество операций, которое может быть выполненно за фиксированный отрезок времени на машине с n процессорами, обратно пропорционально потребности в ресурсе, находящимся в наибольшем дефиците("бутылочное горлышко" программы). Эта потребность в свою очередь обратно пропорциональна времени взаимодействия с этим ресурсом, необходимом для

выполнения программы. Следовательно, в приближении, что основная часть времени уходит именно на взаимодействие с этим ресурсом, мы получаем, что первая и вторая метрика дают один и тот же результат, или S(p) = C(p).

$$C(p) = \frac{X_{max}(p)}{X_{max}(1)} = \frac{D_b(1)}{D_b(p)} = \frac{t_b(1)}{t_b(p)} \cong S(p)$$

Масштабируемость, в зависимости от поведения функции роста производительности, делится на линейную, сублинейную и сверхлинейную. При этом сверхлинейная масштабируемость - не фантастика, если учесть с добавлением процессоров к проекту, добавляется не только вычислительная мощность, но и дополнительная память и прочие ресурсы, что может помочь ускорить выполнение программы сильнее.

По другой классификации маштабируемость можно разделить на масштабируемость с фиксированным объёмом работы и масштабируемость с фиксированным временем работы.

Среди моделей масштабируемости можно выделить четыре: линейная, закон Амдаля, супер-сериальная модель и закон Густафсона, из которых первая описывает линейную масштабируемость, вторая и третья - сублинейную, а последняя - сверхлинейную [1].

ЛИНЕЙНАЯ МОДЕЛЬ

Первая, линейная модель предполагает, что масштабируемость программы может быть выражена с помощью линейной функции. Интересным следствием из определения увеличения масштаба является то, что линейная масштабируемость может иметь в качестве углового коэффициента только единицу. Действительно, если бы угловой коэффициент был бы равен некому k, отличному от единицы. Тогда значение C(1) было бы равно k и не равно единице, чего по определению C(p) быть не может, ведь, согласно ему

$$C(1) = \frac{X_{max}(1)}{X_{max}(1)} = 1$$

Отсюда можно сделать вывод, что сверхлинейная или сублинейная масштабируемость не будет описываться линейной функцией, поэтому к линейной аппроксимации не вполне линейных значений S или P надо относится очень осторожно.

ЗАКОН АМДАЛЯ

Закон Амдаля - модель сублинейной масштабируемости с фиксированным объёмом работы. В ней рассматривается ситуация, когда весь код программы делится на тот, который можно исполнять параллельно, и тот, который необходимо исполнить последовательно. Первая часть кода исполняется на n процессорах в n раз быстрее, но время исполнения второй остаётся неизменной. Если доля второй части кода ненулевая, проявляется ухудшение масштабируемости - она становится сублинейной. Посчитав ускорение по очевидной формуле, а потом произведя несколько эквивалентных преобразований и введя обозначения:

а) σ - доля кода, который не параллелизируется;

б) π - доля кода, который параллелизируется(σ+π=1).

$$S_A = \frac{1}{\frac{t_s}{t_s + t_p} + \frac{t_p}{t_s + t_p}\left(\frac{1}{p}\right)} = \frac{p}{\sigma + \pi / p}$$

получим такую формулу - закон Амдаля:

$$S_A = \frac{p}{1 + \sigma(p - 1)}$$

Из закона Амдаля, максимизируя значение ускорения по количеству процессоров, найдём максимальное возможное ускорение для данной программы. Поскольку функция ускорения в законе Амдаля монотонна по количеству процессоров, то максимум достигается, если устремить его в бесконечность. Отсюда, максимальное значение ускорения будет:

$$\lim_{p \to \infty} S_A = \frac{1}{\sigma}$$

СУПЕРСЕРИЙНА МОДЕЛЬ

Супер-серийная модель является в некотором смысле обобщением закона Амдаля. Это тоже сублинейная модель с фиксированным объёмом работы. Её принципиальное отличие от закона Амдаля в том, что в параллелизуемом участке кода она выделяет долю, уходящую на взаимодействие между процессорами. Это подразумевает, что во время исполнения параллелизуемого кода, процессам нужно обмениваться между собой какой-то информацией. Из-за этого во времени к "параллельному” участку выполнения программы, делящейся на количество процессоров и последовательному участку, не зависящему от него, добавляется ещё слагаемое “время обмена данными с одним процессором”, которое умножается на количество процессоров минус один(нужно посылать сообщения всем, кроме себя). В суперсерийной модели σ означает то же, что и раньше, а под γ понимается доля непараллелизируемого хода, уходящая на обмен информацией между процессорами.

$$C_S(p) = \frac{p}{1 + \sigma[(p-1) + \gamma p(p-1)]}$$

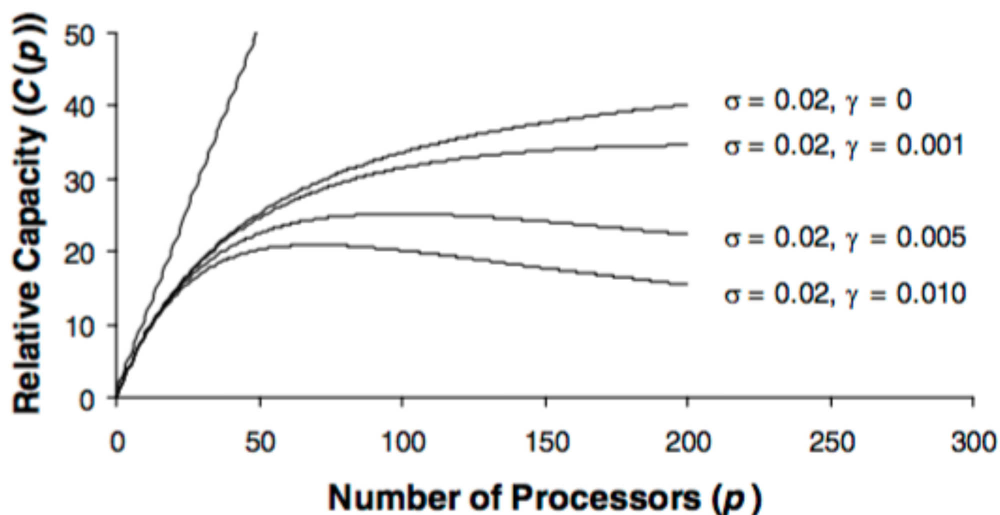
Таким образом, зависимость ускорения от числа процессоров ещё дальше отклоняется от линейной. В отличии от закона Амдаля, здесь функция не является монотонной. Взяв производную, находим, что максимальное значение

достигается при количестве процессоров p_{max}(в случае, если значение выходит нецелым, его, разумеется, следует округлить).

$$C'_S(p) = \frac{1 - \sigma - \sigma\gamma p^2}{(1 + \sigma((p-1) + \gamma p(p-1))^2}$$

$$p_{max} = \sqrt{\frac{1-\sigma}{\gamma\sigma}}$$

Например, при σ = 0,01 и γ = 0,02, p_{max} принимает значение 70, а соответствующее максимальное значение увеличения масштаба будет равнятся приблизительно 20,7962.



ЗАКОН ГУСТАФФСОНА

Последняя из вышеназванных моделей, закон Густафссона принадлежит к другому классу моделей масштабируемости - моделям с фиксированным временем. Модель похожа на закон Амдаля - в ней в коде можно выделить последовательный ts и параллелизируемый tp участки, но поскольку в этот раз мы фиксируем не объём работы, а время работы, уравнение для увеличения масштаба выходит другим. Чтобы выполнить тот же объём работы, что на n

процессорах выполняется за время $t_s + t_p$, для одного процессора понадобится $t_s + p * t_p$.

$$S_G = \frac{t_s + (t_p \times p)}{t_s + t_p}$$

Теперь если обозначить за σ' долю непараллелизируемого кода, выполняющуюся на одном процессоре, получим такое выражение:

$$S_G = p + \sigma'(1 - p)$$

УСКОРЕНИЕ С ОГРАНИЧЕНИЕМ ПО ПАМЯТИ

Последние три модели удобны, однако есть ещё один подход, позволяющий создать модель, обобщающую предыдущие. В ней ограничивающим производительность программы фактором считается память процессоров.

В упрощённой модели, которая не принимает во внимание процессорное взаимодействие [4], мы, как и раньше, моделируем программу разделённой на две части - последовательную и параллельную. Но возростание объёма работы с ростом числа процессоров с собственной памятью описывается с помощью функции g, ставящей объём работы количеству памяти.

делаются два допущения:

1. объём вычислений представим как функция g от имеющейся в наличии памяти;
2. функция g в свою очередь представима как $g(cx) = \bar{g}(c)g(x)$ для всех c и x.

Тогда из равенства W_N = g(M), где W_N - доля параллелизируемой работы на одном процессоре, по второму допущению следует, что

$$W_N^* = g(NM) = \bar{g}(N)g(M)$$

А значит общая формула для увеличения масштаба будет, в итоге:

$$S_N(W^*) = \frac{W_1^* + W_N^*}{W_1^* + W_N^*/N} = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N} W_N}$$

Согласно этой формуле отнесение масштабируемости конкретной программы к линейной, сублинейной или сверхлинейной хависит от функции g(N)(здесь и в дальнейшем имеется в виду подчёркнутое сверху G), зависящей от конкретной программы. Заметим, что при g(N) равной тождественно единице, формула превращается фактически в закон Амдаля. А при g(N) равной N - в закон Густаффсона. Но функция может быть и полиномом степени выше первой. Это зависит от сложности алгоритма программы.

Инструментация

Инструментация - это модификация программы, с помощью внедрения в неё дополнительного кода или оборудования, в целях её анализа. Разумеется, инструментация необходима и при анализе масштабируемости. В [5] разбираются основные виды инструментации.

Поддержка инструментации существует уже на машинном уровне. Практически на всех машинах выпускаемых в наше время установлены некоторые часы, которые могут быть использованы для измерения времени выполнения программ с точностью до миллисекунд, а также - на более высоком уровне, регистры, отсчитывающие время - они дают микро- и наносекундную точность. Однако, чтобы использовать эти возможности машины для измерений масштабируемости, надо дать ей во время выполнения измеряемой программы нужные для засечения выполнения определённых операций, инструкции. Инструментация, засекающая длительность выполнения определённых частей программы или ведущая учёт, случилось ли событие определённого типа называется инструментация отслеживания.

Инструментация может производиться на разных этапах - на уровне исходного кода, бинарника и компоновщика.

В первом методе программистом вносится изменения в исходный код, вызывающий инструментационные команды. Здесь модификация кода производится до начала компиляции и запуска программы, то есть это статический метод. Плюсом метода являются его простота, минусом то, что его можно использовать только для приложений, исходный код которых тебе представлен.

Во втором методе модифицируется код объектного файла напрямую. Как правило в начале отслеживаемого добавляется безусловный переход на инструментационный код, по завершению исполнения которого, анализируемая программа продолжает выполнятся дальше. У бинарной инструментации больше возможностей - она может быть исполнена как статически, перед запуском программы, так и динамически, в процессе её выполнения. Также, здесь нам не обязательно иметь исходник программы, чтобы использовать этот метод, хотя и необходимость в возможности изменять объектный файл тоже накладывает свои ограничения - например, как правило, мы не можем применять этот метод к функциям ядра системы.

Последний метод инструментации основан на том, что использование любой сторонней библиотеки в программе происходит с помощью компоновщика. Если сделать для определённой библиотеки обёртку, которая кроме вызова функций исходы библиотеки будет совершать инструментацию и настроить компоновщик, чтобы он связал приложение с новой библиотекой вместо старой, мы достигнем цели не поменяв ничего в коде программы. Этот метод удобен, когда нужно проанализировать взаимодействие программы с определённой библиотекой. В этом методе может использоваться как статический, так и динамический компоновщик. В первом случае, перед запуском программы необхоимо произвести перекомпоновку, что не обязательно делать во втором случае.

Помимо отслеживания также используют другой подход - выборку. Это вид асинхронной инструментации, в котором мы совершаем периодические измерения, из которых, в зависимости от их характера, можем установить, на что программа тратит время или как использует определённый ресурс.

Любая инструментация имеет нежелательный эффект - на исполнение инструментационных функций требуется какое-то время и процессорная мощность. Это несколько искажает сделанные с их помощью измерения. В отслеживании чем чаще исполняется в коде замеряемая функция, тем больший процент времени уходит на исполнение инструментационного кода и тем сильнее искажение. В инструментации выборкой нужно ответственно подходить к выбору частоты, с которой проводить измерения. Если проводить их слишком часто, это сильно повлияет на производительность программы, но слишком редкое проведение измерений, в свою очередь, снизит их репрезентативность.

Advanced NSOperations

Класс NSOperation языка программирования Swift - абстрактный класс, созданный для инкапсуляции кода и данных. Каждый объект класса выполняет операцию только один раз. Операции исполняются в очереди - объекте класса NSOperationQueue, которая в свою очередь, является более высокоуровневой обёрткой к функциональности библиотеки Grand Central Dispatch на языке C.

В подклассе Operation абстрактного класса NSOperation, представленном на WWDC в 2015 году, возможности NSOperation были расширены. У объекта этого класса такой жизненный цикл, связанный с положением операции в очереди: только созданный и положенный в определённую очередь, объект находится в состоянии pending; далее проделывается проверка выполнения всех условий, необходимым для исполнения данной операции - среди условий в том числе может быть и необходимость завершения выполнения других операций - и только после того, как все условия оказываются выполнены, состояние операции меняется на ready; теоретически, сразу после этого операция начать исполняться, однако, на практике ей может потребоваться дождаться, пока очередь закончит выполнение текущей операции; и наконец, после своего завершения, операция переходит в состояние finished. Кроме того, в любой момент, операция может быть отменена - перейти в состояние cancelled.

Таким образом, с помощью проверки выполнения условий для каждой операции, мы убеждаемся, мы частично освобождаемся от блокирующих операций, что соответствует идеологии асинхронного программирования.

Кроме этого, с помощью структуры BlockObserver к определённым событиям жизненного цикла любой операции можно привязывать блоки кода, что является прямым аналогом функций обратного вызова в языках вроде javascript.

Отчёт о проделанной работе

Задача программы

Задачей созданной много инструмента для разработчика является тестирование производительности на машинах с разным количеством процессоров и анализ полученных данных на основе известных моделей масштабируемости.

Конкретизируя, она предназначена измерять, сколько операций выполняется данной программой за определённое время на машинах с одним и больше количеством процессоров(всего не менее трёх измерений всего, по одному на каждой машине). Потом измеренные числа подаются в элемент класса Experiment, который сохраняет только производительности относительно однопроцессорной. Далее для модели, основанной на законе Амдаля, Густаффсона, и суперсерийной модели ищутся наиболее соответствующие имеющимся парам "количество процессоров, производительность" значения параметров моделей.

Детали реализации

Для закона Амдаля и Густаффсона ищется только один параметр - σ, доля инструкций, исполняемых параллельно, и σ', то же самое, но случае выполнения программы на одном процессоре. Для суперсерийной модели - два параметра: σ, доля инструкций, исполняемых параллельно, и γ, доля в них отвечающих за межпроцессорную коммуникацию.

Для каждой из этих моделей поиск происходит по одинаковому алгоритму. Из формулы модели для увеличения масштаба вычитается получившееся в эксперименте увеличение масштаба. Подобное проделывается для всех имеющихся экспериментов. Используется квадратичная функция потерь, то есть суммируются квадраты всех этих разностей. Полученную в результате этого функцию мы минимизируем на кубе [0,1] в пространстве с размерностью количества искомых параметров. Выбор именно такого

множества объясняетя тем, что искомые параметры имеют значение части от целого, значит, не могут превосходить 1.

Полученная задача - задача условной оптимизации. Для решения её был выбран метод проекции градиента. Его использование оправдано, так как множество, на котором мы ищем решения замкнуто и выпукло.

В процессе применения модели ускорения с ограничением по памяти, тем же методом обойтись не получится, так как в ней одним из неизвестных параметров является функция. Чтобы подобрать её нужно использовать какой-то из методов нелинейной регрессии. Относительно функции g(N) было решено сделать предположение, что она является полиномом невысокой степени. Далее, было замечено, что из формулы выносится выражение

$$\frac{\sigma}{\sigma - 1} \cdot \overline{g}$$

Поэтому был избран путь сначала найти значение этого выражение с помощью метода Ньютона-Рафсона(а именно найти коэффициенты при степенях в искомом многочлене), а потом, когда останется только один неизвестный параметр σ, найти его тем же способом, что и в предыдущих случаях.

Инструментация

Для анализа масштабируемости приложений, использующих Advanced NSOperation было решено использовать инструментацию на уровне кода. Программа запускает программу с изменённым кодом, которая с момента начала работы программы засекает 30 минут и считает общее количество объектов класса или подклассов Operation, достигших состояния finished, в течении этого времени. После чего он шлёт результат на локальный сервер.

Заключение

С помощью созданного программного обеспечения можно легко тестировать масштабируемость приложения, написанного с использованием класса Operation и получать простую для понимания интерпретацию её значения. Это может помочь разработчикам лучше понимать и рассчитывать производительность своего кода.

В дальнейшем проект можно развивать в множестве направлений: добавить новые, удобные в специальных случаях модели масштабируемости, расширить класс тестируемых приложений, добавить статический анализатор кода приложения, который делал бы предположения, какие конкретно места могли вызвать проблемы с масштабируемостью.

Литература

1. Lloyd G. Williams, Connie U. Smith. 2004. Web-application scalability. A model-based approach. Software engineering research and performance engineering services.

2. В. Г. Жадан. 2015. Методы оптимизации. Часть 2. Численные методы. Москва, МФТИ.

3. Дж. Рихтер. 2008. Windows для профессионалов: создание эффективных Win32 приложений с уче том специфики 64 разрядной версии Windows / Пер. с англ. — 4 е изд. — Спб.: Пи тер; М.: Издательство «Русская Редакция»; 2008.

4. Xian-He Sun and Lionel M. Ni. 1993. Scalable problems and memory-bounded speedup. Journal of parallel and distributed computing. 19:27-37

5. Todd Gamblin. 2009. Scalable performance measurements and analysis. Chapel Hill.

6. Maurice Herlihy, Nir Shavit. 2008. The art of multiprocessor programming. Morgan Kaufmann Publishers.

7. R. Jain. 1990. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, New York, NY, John Wiley, 1990. The Art of Computer

Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, New York, NY, John Wiley.