ИМИТАЦИОННАЯ МОДЕЛЬ ПУЛА ПОТОКОВ ДЛЯ СЕРВЕРА БАЗ ДАННЫХ

И.И. Труб, А.А. Копытов (Москва), А.А. Строганов (Днепр, Украина)

Введение

Имитационное моделирование используется для эффективного принятия решений не только в прикладных предметных областях, таких как различные отрасли народного хозяйства и военное дело, но также и в проектах, связанных с разработкой программного обеспечения и проектированием сложных IT-систем. Одной из таких систем является пул потоков (трэдпул), активно применяемых в самых разнообразных программных продуктах на протяжении последних примерно 25 лет. Данная технология появилась как альтернатива концепции «одно соединение – один поток», так как позволяет во многих случаях не только экономить ресурсы, но и повысить производительность программного продукта в целом. Она заключается в переиспользовании уже существующего потока для обработки следующей задачи. Существует множество различных реализаций трэдпула, наиболее подробный обзор которых приведен в [9]. По отдельным реализациям имеется большое количество материалов. Так, один из первых трэдпулов для брокера объектных запросов CORBA описан в [13], трэдпул для Android – в [1], для сервера приложений Oracle GlassFish – в [10]. Поучительный пример эффективного использования модели трэдпула в Python для решения сложной научной задачи приведен в [23]. Трэдпул от компании Microsoft для CLR описан в фундаментальном труде [16], а последняя реализация доступна в [20]. Собственное расширение модели потоков в Java в виде трэдпула разработано в отечественном академическом исследовании [21]. Большое внимание уделяют этому вопросу также разработчики известных СУБД, в частности, MySQL [14], MariaDB[18], Percona[12]. Особенностями подхода, использованного в этих реализациях, является разделение всего трэдпула на группы потоков, где каждая группа имеет свои упорядоченные по приоритетам очереди запросов на обслуживание и свои подгруппы активных, ожидающих и свободных потоков; отдельный поток-таймер в таком трэдпуле отслеживает застрявшие потоки и запросы. Забегая вперед, отметим, что модель именно такой разновидности трэдпула рассматривается в данной работе.

Все трэдпулы характеризуются большим количеством параметров, которые назначаются проектировщиком и оказывают влияние на итоговую эффективность трэдпула. Основной из них – размер трэдпула, который задает ширину параллелизма (*concurrency level*), в качестве которой, например, для упомянутых выше СУБД-реализаций является количество групп. Этот выбор определяется множеством факторов, такими как общая нагрузка на сервер и профиль этой нагрузки (*workload*). Так, для CPU-ориентированной (*CPU-bound*) нагрузки и нагрузки, ориентированной на ввод-вывод (*IO-bound*), даже при одном и том же количестве одновременных соединений, оптимальные значения размера трэдпула могут коренным образом различаться. Именно поэтому большое число работ содержит в том или ином виде рекомендации, как же выбирать этот параметр. Назначать его можно статически – как постоянную величину, так и динамически, постоянно подстраивая под меняющуюся нагрузку. Статический подход рассмотрен в работах [1,7,8,10,11], где приведены различные вариации формул, опирающихся на доступное число процессоров, среднее время обслуживания запроса на CPU и среднее время простоя CPU из-за ввода-вывода, а также на известный в теории очередей закон Литтла. Что же касается алгоритмов динамического подхода, то здесь наибольших результатов добилась Microsoft. В получивших широкую известность работах [5, 6] рассмотрено применение известного в теории метода *HillClimbing* для оптимизации размера трэдпула, практические же результаты этого применения описаны в [19]. В [4] описана, а в [20] реализована

вариация *HillClimbing*-подхода, основанная не на вычислении градиента, а на преобразованиях Фурье, как обладающая большей устойчивостью к случайным воздействиям. Другие теоретические подходы к решению задачи динамического выбора размера трэдпула предложены в [9] и [17]. Кроме того, [17], опубликованная в 2021 году, содержит полную и актуальную библиографию вопроса.

Вместе с тем, дать более полное и глубокое представление о работе такой сложной системы как трэдпул может помочь имитационная модель. Итоговая производительность (выраженная, например, в среднем числе обслуженных запросов в единицу времени) зависит не только от размера трэдпула, но и от других параметров. Чтобы установить значимость их влияния, требуется большое количество длительных и дорогих натурных экспериментов на рабочих серверах, а хорошо спроектированная имитационная модель позволит сделать это гораздо быстрее, собственно, это и есть ее основное достоинство для любых задач. Этот подход к исследованию трэдпула на данный момент используется еще довольно редко. Можно отметить работу [2] (где использован специфический и малодоступный инструментарий) и недавнюю работу украинских специалистов [15], где применены стохастические сети Петри. Подход к моделированию и оптимизации трэдпула с помощью сетей Петри используется также в [22]. В данной работе предлагается имитационная модель трэдпула, где за основу моделируемой системы взята реализация, описанная в [12]. Сама модель реализована на языке C++ с помощью схемы, описанной в [24], позволяющей гибко учесть все алгоритмические нюансы системы в полном объеме. В разделе II описан сам трэдпул, в разделе III – архитектура предложенной модели, в разделе IV – некоторые результаты ее валидации, в разделе V – выводы и рекомендации по ее применению.

I. Описание моделируемого трэдпула

Если опустить вторичные детали, граф вызовов функций трэдпула выглядит так, как показано на рис. 1:

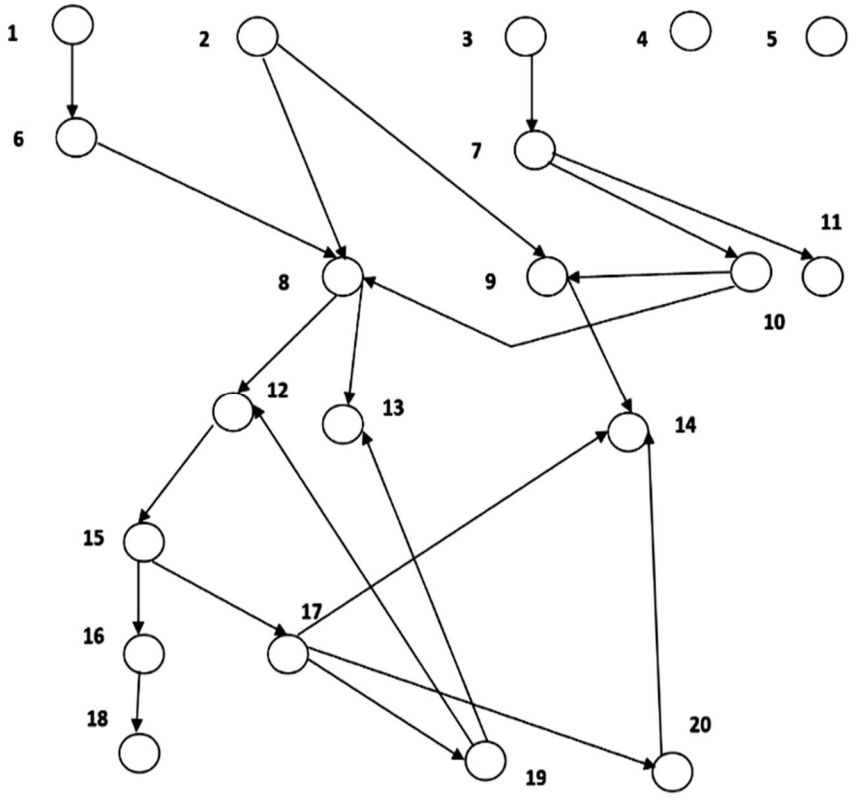

Рис. 1. Граф вызовов функций трэдпула. Обозначения соответствуют функциям, приведенным в таблице 1

Таблица 1. Описание функций трэдпула

1 add_connection	Добавляет новое соединение, выбирает для него группу потоков	11 timeout_check	Проверяет, не истек ли тайм-аут для обработки запроса; удаляет соединение, если истек
2 wait_begin	Коллбэк для начала простоя потока из-за I/O	12 create_worker	Создает новый поток
3 start_timer	Запуск потока-таймера для отслеживания застрявших потоков	13 wake_thread	Будит неактивный поток
4 set_tp_size	Задает размер трэдпула	14 too_many_threads	Проверяет, не слишком ли много в группе активных потоков
5 wait_end	Коллбэк для окончания простоя потока	15 worker_main	Основная функция для потока из трэдпула
6 queue_put	Запись нового соединения в очередь	16 handle_event	Подготовка к выполнению запроса
7 timer_thread	Основная функция для потока-таймера	17 get_event	Назначает соединение готовому к работе потоку (дает ему работу)
8 wakeCreateThread	Создает новый поток или будит неактивный	18 process_request	Выполнение запроса потоком
9 queues_are_empty	Проверяет, пусты ли очереди	19 listener	Поток-поллинг, повторная выборка соединений на дескрипторе группы
10 check_stall	Обрабатывает застрявшие потоки	20 queue_get	Выборка соединения из очереди

II. Описание имитационной модели

Перечислим случайные величины, являющиеся входными для модели и генерируемые датчиком случайных чисел в соответствии с выбранным распределением:

· входной поток соединений – распределение интервалов времени между вызовами функции *add_connection*;

· длительность создания нового потока – тайминг для функции *create_worker*;

· длительность одного раунда активности потока: от начала обработки запроса до первого вызова *wait_begin*, либо между вызовами *wait_end* и *wait_begin*, либо между *wait_end* и завершением обработки запроса;

· длительность одного раунда простоя CPU: между вызовами *wait_begin* и *wait_end*;

· количество активных раундов за время обслуживания одного запроса;

· время между завершением обслуживания запроса и выбором того же персистентного соединения процедурой поллинга для назначения ему нового потока и обслуживания следующего запроса.

Выходные значения модели – среднее число обслуженных запросов в секунду (*queries per second*) и средняя длительность обслуживания запроса (*latency*).

Модель реализована на следующих предметных классах: *Threadpool* (синглетон), *Threadgroup*, *Connection*, *Thread*, *Timer*(синглетон). Для экземпляров класса *Thread* установлены состояния:

· *creating* – создание потока;

· *active* – обслуживание запроса;

· *waiting* – ожидание ввода-вывода;

· *idle* – простой потока, когда запрос обслужен, а новый еще не назначен;

· *polling* – в этом состоянии может находиться только поток, занятый поллингом (*listener*).

Для экземпляров класса *Connection* установлены состояния:

· *in usual queue* – соединение ожидает назначения потока в обычной очереди;

· *in prio queue* – соединение ожидает назначения потока в льготной очереди;

· *threading* – соединению назначен поток, запрос обслуживается;

· *between* – запрос обслужен, соединение ожидает выборки потоком-*listener*'ом.

Возможные переходы между состояниями показаны на рис. 2 и 3.

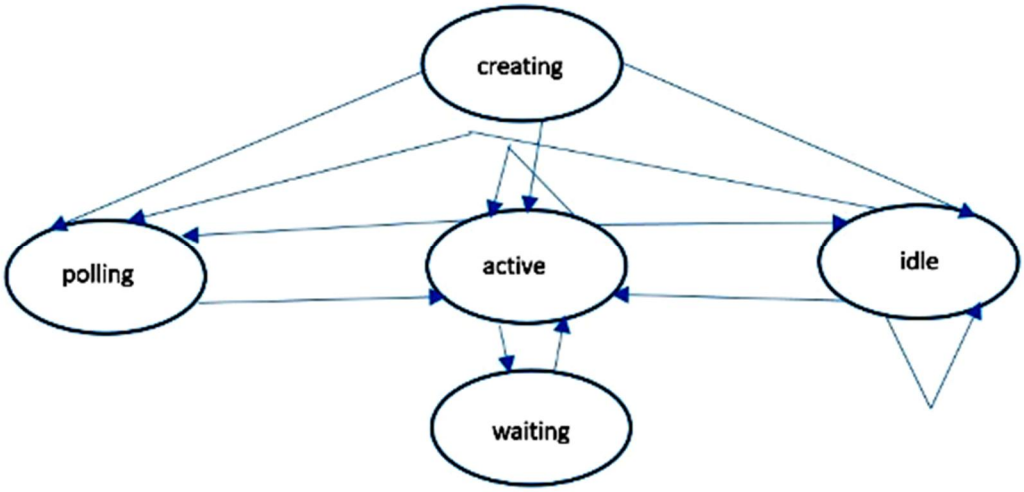

Рис.2. Модель состояний для класса Thread

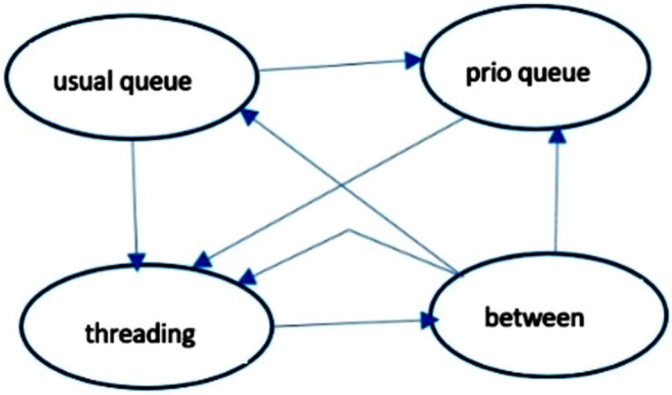

Рис. 3. Модель состояний для класса Connection

Параметрами, которые исследователь может варьировать для их оптимизации, помимо размера трэдпула (количества групп), являются следующие свойства класса *Threadgroup*:

· *oversubscribe* – максимальное количество активных потоков в группе (по умолчанию – 3);

· *queue_put_limit* – создать новый поток внутри функции *queue_put*, если число активных потоков в группе меньше данного значения (по умолчанию – 0);

· *create_thread_limit* – не создавать поток в функции *create_worker*, если число активных потоков в группе превышает данное значение (по умолчанию – 1);

· *listener_wake_limit* – не будить *idle*-поток в потоке *listener*, если число активных потоков в группе превышает данное значение (по умолчанию – 0);

· *listener_create_limit* – не создавать новый поток в потоке *listener*, если число активных потоков в группе превышает данное значение (по умолчанию – 1).

Структура модели в виде графа взаимных вызовов методов классов приведена на рис. 4. Названия методов даны в табл. 2. Основной цикл модели представлен на листинге 1.

Отдельно остановимся на вопросе, каким образом моделируются затраты времени CPU на переключение контекста процесса, так как именно по этой причине увеличение числа групп в случае *IO-bound* нагрузки с некоторого момента приводит к деградации.

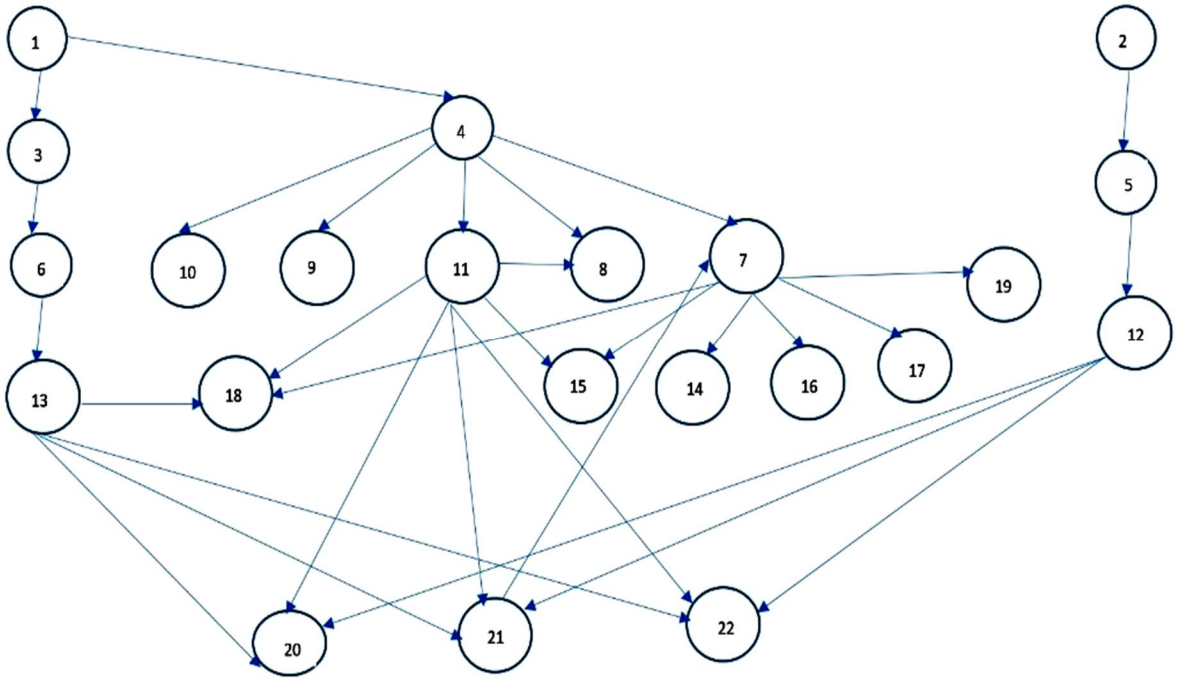

Рис. 4. Структура имитационной модели

Таблица 2. Методы классов имитационной модели

1. Threadpool::run	12. Threadgroup::check_stall
2. Timer::run	13. Threadgroup::queue_put
3. Threadpool::add_connection	14. Threadgroup::queue_get
4. Threadgroup::run	15. Connection::to_threading
5. Threadpool::check_stall	16. Thread::to_polling
6. Threadgroup::add_connection	17. Threadgroup::get_connection_from_polling
7. Threadgroup::assign_connection_to_thread	18. Connection::to_usual_queue
8. Thread::to_active	19. Thread::to_idle
9. Thread::to_waiting	20. Connection::to_prio_queue
10. Connection::to_between	21. Threadgroup::wake_thread
11. Threadgroup::listener	22. Threadgroup::create_worker

Листинг 1.

```
int main() {
Threadpool *tpl = Threadpool::getInstance();
Timer *tmr = Timer::getInstance();
srand((unsigned)time(0));
for (unsigned long i=0; i<60000000;i++) {
        Tpl->run();
        Tmr->run();
}
}
```

Пусть N – число активных потоков, M – число CPU, причем $M < N$, a – время переключения контекста (параметр модели), t – время обслуживания запроса (в терминах теории очередей – длина заявки). Тогда за 1 такт модельное время за всех заявок продвинется не на 1, а на величину $M/N-a$, N заявок выполнятся за время $(tN)/(M-aN)$ и производительность равна $(M-aN)/t$ заявок в единицу времени, т. е. с ростом N она действительно снижается. Таким образом, формулируем правило: при выполнении условия

$$M - aN > \frac{N}{\left\lceil \frac{N}{M} \right\rceil} \qquad (1)$$

для всех N активных потоков остаточная длина заявки уменьшается на величину $M/N-a$, в противном же случае действуем так: у произвольно взятых M из N активных потоков остаточную длину уменьшаем на единицу, у остальных $N-M$ остаточная длина не изменяется. Второй случай соответствует ситуации, когда время переключения контекста слишком велико и использовать дисциплину разделения процессора не имеет смысла. Усеченные квадратные скобки в формуле (1) означают деление с округлением вверх.

III. Проверка модели и результаты

Валидность модели проверялась на контрольных примерах, где выходные результаты модели (производительность и средняя длительность исполнения запроса) сравнивались при той же моделируемой нагрузке с результатами, полученными на реальном сервере с помощью известной утилиты тестирования производительности MySQL *sysbench*. Модель показала расхождение с данными *sysbench* по обоим выходным параметрам в пределах 2%. В этом разделе мы рассмотрим результаты работы модели для двух принципиально различных профилей нагрузки – *CPU-bound* и *IO-bound*. Вот примеры различия во входных данных. На рис. 5 и 6 показаны гистограммы для собранных данных (около 1000 наблюдений) для длительности одного раунда ожидания CPU в микросекундах (иными словами, нахождения потока в состоянии *waiting*). В первом случае моделировались 1024 персистентных соединений с сервером при 32 ядрах, во втором случае – 128 соединений с более трудоемкими запросами при 64 ядрах. Продолжительность моделирования – 60 млн. тактов (микросекунд), т. е. 1 минута. Из рисунков видно, что длительность ожидания ввиду гораздо более значительного удельного веса операций ввода-вывода в запросах во втором случае значительно выше. Это означает бОльшее время простоя процессора и соответственно возможность эффективной установки размера трэдпула выше количества физических CPU.

На рис. 7 и 8 показаны зависимости производительности от размера трэдпула. Эти графики хорошо соответствуют известным паттернам такой зависимости, классифицированным, например, в [3]. Из графика видно, что при *CPU-bound* нагрузке

размер трэдпула выше числа CPU не дает эффекта – процессоры и так постоянно загружены. Поэтому здесь основная цель модели – не столько максимизировать выход, сколько минимизировать размер трэдпула. Для *IO-bound* нагрузки картина иная – производительность продолжает расти и после значения 64, достигая максимума при размере около 185, затем начинает снижаться из-за конкуренции потоков на CPU и затрат на смену контекста процесса.

IV. Выводы

Итак, где и каким образом можно применить построенную модель трэдпула – как для описанной в этой работе реализации, так и для любой другой:

1) выявление настроек и локальных алгоритмических решений, к изменению которых производительность трэдпула наиболее чувствительна, и формирование на этой основе рекомендаций для инженера-проектировщика и администратора СУБД;

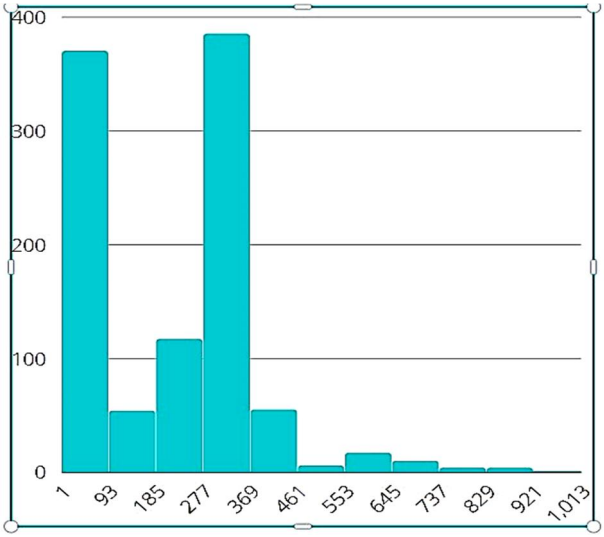

Рис 5. Длительность wait-раунда для CPU-bound профиля

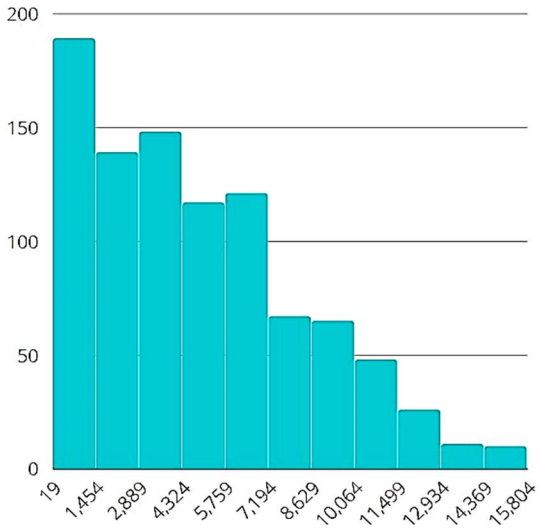

Рис 6. Длительность wait-раунда для IO-bound профиля

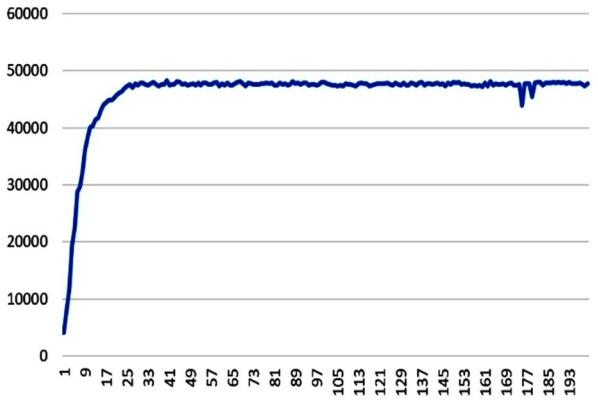

Рис 7. “concurrency level – throughput” для CPU-bound

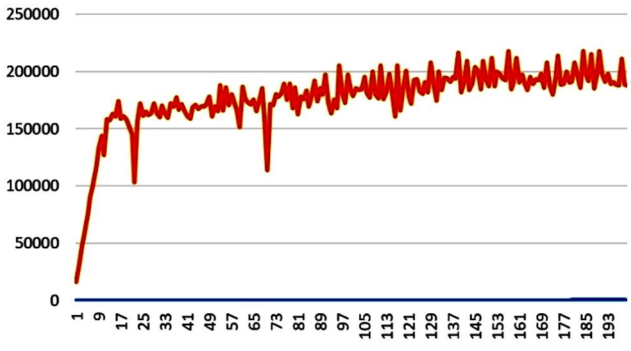

Рис 8. “concurrency level – throughput” для IO-bound

2) выявление зависимости выходных величин от вида и параметров входных распределений;

3) поиск оптимальных значений настроек и выявление их зависимости от количественных и качественных показателей нагрузки на сервер;

4) динамическая оптимизация настроек трэдпула: по ходу работы трэдпула собирается и обрабатывается статистика, на основании которой периодически запускается модель для быстрого поиска оптимальных значений наиболее важных настроек.

Литература

1. Better performance through threading. – developer.android.com/topic/performance/threads.
2. **Boer F.S., Grabe I., Jaghoori M.M., Stam A., Yi W.** Modeling and Analysis of Thread-Pools in an Industrial Communication Platform // ICFEM'09: Proceedings of the 11-th International Conference on Formal Engineering Methods, November 2009, pp.367-386.
3. **Dongping X.** Performance study and dynamic optimization design for threadpool systems (2004). – digital.library.unt.edu/ark:/67531/metadc780878/m2/1/high_red_d/835380.pdf.
4. **Fuentes E.** Concurrency – Throttling Concurrency in the CLR 4.0 Threadpool (September 2010). – docs.microsoft.com/en-us/archive/msdn-magazine/2010/September/concurrency-throttling-concurrency-in-the-clr-4-0-threadpool.
5. **Hellerstein J.L., Morrison V., Eilebrecht E.** Applying Control Theory in the Real World // ACM'SIGMETRICS Performance Evaluation Rev. 2009. Vol. 37, Issue 3. P. 38-42.
6. **Hellerstein J.L., Morrison V., Eilebrecht E.** Optimizing Concurrency Levels in the .NET Threadpool. – FeBID Workshop 2008, Annapolis, MD USA.
7. **Ilinchik A.** How to set an ideal thread pool size (April 2019). – engineering.zalando.com/posts/2019/04/how-to-set-an-ideal-thread-pool-size.html.
8. Java Concurrency in lock optimization and optimization thread pool. – programmersought.com/article/84012626442.
9. **Nazeer S., Bahadur F.** Prediction and Frequency Based Dynamic Thread Pool System // Int. J. of Comp. Sci. and Information Security. 2016. Vol. 14, No. 5. P. 299 – 308.
10. Oracle® GlassFish Server 3.1 Performance Tuning Guide. – https://docs.oracle.com/cd/E18930_01/ pdf/821-2431.pdf.
11. **Pepperdine K.** Tuning the Size of Your Thread Pool (May, 2013) – infoq.com/articles/Java-Thread-Pool-Performance-Tuning
12. Percona Server for MySQL: Thread Pool. – https://www.percona.com/doc/percona-server/5.7/ performance/threadpool.html.
13. **Pyarali I., Spivak M., Cytron R.** Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA // ACM'SIGPLAN Notices. 2001. Vol. 36, Issue 8, P. 214-222.
14. **Ronstrom M.** MySQL Thread Pool: Summary (October 2011). – http://mikaelronstrom.blogspot.com/ 2011/10/mysql-thread-pool-summary.html.
15. **Stetsenko I., Dyfuchyna O.** Thread Pool Parameters Tuning Using Simulation // Adv. in Comp. Sci. for Engineering and Education II (ed. Hu Z.), Springer. 2020. P. 78 – 89.
16. **Terrell R.** Concurrency in .NET: Modern patterns of concurrent and parallel programming. – Simon and Schuster Publishing House, 2018, 568 pp.
17. **Timm J.** An OS-level adaptive thread pool scheme for I/O-heavy workloads. – Master thesis, Delft University of Technology, 2021 (repository.tudelft.nl)
18. Thread Pool in MariaDB. – mariadb.com/kb/en/thread-pool-in-mariadb
19. **Warren M.** The CLR Thread Pool 'Thread Injection' Algorithm (April 2017). – codeproject.com/Articles/1182012/The-CLR-Thread-Pool-Thread-Injection-Algorithm.
20. github.com/dotnet/coreclr/blob/master/src/vm/win32threadpool.cpp.
21. **Акопян М.С.** Расширение модели ParJava для случая кластеров с многоядерными узлами // Труды ИСП РАН. 2012. Том 23.
22. **Бабичев С.Л., Коньков К.А., Коньков А.К.** Использование пула вычислительных потоков со статическим планированием // Труды МФТИ. 2012. Т.4, № 3. С. 162-170.

23. **Клячин В.А.** Реализация параллельного алгоритма геометрического хеширования на основе пакета NumPy и пула процессов // Вестн. Волгогр. Гос. Ун-та. Сер. 1. Мат.-Физ. 2015. Вып. 4 (29). С. 13-23.
24. **Труб И.И.** Объектно-ориентированное моделирование на C++. СПб, 2005. 416 с.