DOI 10.37882/2223–2966.2021.05–2.06

ПРОФАЙЛЕРЫ И МЕХАНИЗМЫ ОПТИМИЗАЦИИ В ПРОГРАММИРОВАНИИ

PROFILERS AND OPTIMIZATION MECHANISMS IN PROGRAMMING

A. Bazarova

Summary. The article is devoted to the consideration of profilers and optimization mechanisms in programming. The research process covered server-side profiling, desktop-side profiling, and also hybrid profiling. Retrace is considered as an example of a profiler on the server side, its capabilities are also described in detail. Special attention is paid to the hybrid profiling solution for this purpose, the Prefix profiler is considered. Also, special emphasis is placed on the AQtime profiler, which has a very wide functionality and allows you to perform performance profiling, debugging memory and resources for compilers Microsoft, Borland, Java, Intel, Compaq and GNU. Extended methods are considered as programming optimization mechanisms, namely: Hill Climbing, Simulated Annealing, Genetic Algorithm. Their capabilities and disadvantages are indicated.

Keywords: profiling, programming, optimization, code, method.

Базарова Анна Максимовна
Старший преподаватель, Ухтинский государственный технический университет
anna_sh94@inbox.ru

Аннотация. Статья посвящена изучению профайлеров и механизмов оптимизации в программировании. В процессе исследования рассмотрено профилирование на стороне сервера, на стороне рабочего стола, и также гибридное профилирование. В качестве примера профилировщика на стороне сервера представлен Retrace, также детально описаны его возможности. Отдельное внимание уделено решению гибридного профилирования, с этой целью проанализированы возможности профилировщика Prefix. Также особый акцент сделан на профилировщике AQtime, который имеет очень широкий функционал и позволяет проводить профилирование производительности, отладку памяти и ресурсов для компиляторов Microsoft, Borland, Java, Intel, Compaq и GNU. В качестве механизмов оптимизации программирования представлены расширенные методы, а именно: метод Hill Climbing, Simulated Annealing, генетический алгоритм. Обозначены их возможности и недостатки.

Ключевые слова: профилирование, программирование, оптимизация, код, метод.

При разработке большой программной системы возникает вопрос выбора платформы для проведения необходимых действий и операций. Основными критериями в процессе выбора является наличие инфраструктуры для разработчиков программного обеспечения и ряда готовых функций для решения типовых задач. Но основным критерием остается надежность и быстродействие [1]. Высокая сложность и значительный размер современных программных систем побуждают к разработке и применению новых методов выявления имеющихся дефектов и проведения глубокого анализа.

Потребность в оценке качества программного обеспечения (ПО) возникает по двум главным причинам: в критически важных сферах отказ ПО приводит не только к материальным потерям, но и к ущербу для людей; применение качественного ПО требует меньше средств на этапе сопровождения, включая использование, поддержку и внесение изменений для неотложных нужд. Если вторая причина приводит к значительным финансовым убыткам, а также увеличению нагрузки на команду разработчиков, что можно отнести к классу значительных последствий, но не критических, то недостатки в программном обеспечении, которые наносят вред человеческому здоровью, являются недопустимыми.

Например, в феврале 2007 г. двенадцать истребителей F22 выполняли перелет с военной базы США на Гавайях в Японию. В момент пересечения временной границы на всех самолетах по причине программной ошибки отказали бортовые компьютеры [2]. В 2009 г. широкую огласку получила информация о возможности перехвата радиообмена беспилотного летательного аппарата ВВС США MQ-1. Допущенные недостатки в проектировании программной системы аппарата привели к отсутствию шифрования потока его данных, что открыло беспрепятственный доступ к конфиденциальной, критически важной информации. В настоящее время существуют специализированные приложения и методы, которые на разных этапах жизненного цикла помогают анализировать и оценивать качество ПО, однако они лишь частично выполняют задачи обеспечения качества программ.

В данном контексте особую актуальность и важность приобретает динамический анализ ПО, методы и механизмы оптимизации программирования. Динамический анализ представляет собой метод анализа, в рамках которого происходит непосредственный запуск программы на реальном или виртуальном процессоре в обычном режиме или в режиме отладки. Динамический анализ программных средств также может служить как допол-

нительный инструмент для проверки, действительно ли найденные при статическим исследовании недостатки являются уязвимостями.

Одним из наиболее эффективных инструментов динамического анализа является профилирование, которое измеряет, например, пространство (память) или временную сложность программы, эффективность использования определенных инструкций, частоту и продолжительность вызовов функций. Чаще всего профилирующая информация помогает оптимизировать программу. Следует отметить, что задачи профилирования и пути их решения могут быть обобщены для определения профилей метрик, которые используются для оценки качества ПО, профилей дефектов.

Профилирование выполняется с помощью специальных программных средств, называемых профайлерами. Инструменты программного анализа критически важны для понимания поведения программы. Компьютерным архитекторам необходимы такие инструменты, чтобы оценить, как программы будут выполняться на новой архитектуре. Разработчики программного обеспечения также нуждаются в профайлерах, чтобы проанализировать программы и идентифицировать критические части кода. Авторы компиляторов часто используют такие методы, чтобы выяснить, как хорошо выполняется их планирование инструкций или алгоритм предсказания. Выходящим результатом использования профайлера является поток записанных событий или статистический краткий отчет наблюдаемых явлений. Профайлеры используют широкое разнообразие методов, чтобы собрать данные, в том числе аппаратные прерывания.

Однако на сегодняшний день недостатками известных методов профилирования является их неполная формализация, отсутствие унифицированных процедур проведения тестов на всех этапах жизненного цикла кода, несовершенство соответствующих моделей и информационных технологий оценки качества ПО.

Таким образом, указанные обстоятельства предопределяют выбор темы проводимого исследования, а также подтверждают его теоретическую и практическую значимость.

На сегодняшний день уже есть ряд научных исследований, которые посвящены разработке и совершенствованию методов оценки качества ПО. Из числа наиболее известных авторов можно выделить Brandtner, M., Galdi, V.; Ippolito, L.; Piccolo, A.; Silvester, P.P.; Полякову Э.Р., Шустова А.Л., Придачкина Д.Г., Долгова Е.П., Новикова С.В.

Описанию средств оптимизации и моделей динамической компиляции посвящены труды Khoshgoftaar, T.M.; Xiao, Y.; Gao, K.; Srivastava, P.R.; Kumar, K.; Prasad, S.K.; Бождай А.С., Евсеевой Ю.И., Гудкова А.А.

Вопросами повышения полноты оценки качества ПО на основе разработки и практического применения методов и инструментальных средств, основанных на технологиях профилирования, занимаются Johnson, G.L.; Xu, X; Braccini, G.; Fabbrini, F.; Fusani, M.; Pappas, C.; Грингауз Т.К., Онин А.Н., Лазеба М.А., Кулагин И.И., Курносов М.Г.

Однако недостаточное качество процессов профилирования обусловливает наличие скрытых дефектов в ПО. Это представляет особую опасность, поскольку они трудно выявляются и существенно влияют на конечные характеристики работы программ.

С учетом вышеизложенного, цель статьи заключается в рассмотрении современных профайлеров, выявлении их возможностей и ограничений, а также изучении механизмов оптимизации в программировании.

Как известно, оптимизировать код достаточно сложно, это требует времени и проведения ряда исследований от разработчиков. Без надлежащих инструментов программисты вынуждены прибегать к более медленным и менее эффективным способам оптимизации своих приложений. Некоторые разработчики используют «предварительную оптимизацию» кода, они пытаются угадать, где могут возникнуть проблемы с производительностью, и реорганизовать свой код, пытаясь устранить проблемы до того, как они появятся. Такой подход проблематичен, потому что разработчик часто неправильно диагностирует потенциальное узкое место. Он смотрит только на свой собственный код, а не на полную базу кода, таким образом упуская проблемы с интеграцией. Кроме того, программист может не иметь представления об ожидаемом поведении своих целевых пользователей или сосредотачивает внимание на редко используемой области кода.

В тоже время, несмотря на очевидные преимущества, программисты не спешат внедрять инструменты профилирования. Они жалуются, что традиционные профилировщики агрессивны и сложны в использовании. В некоторых случаях программисты обнаружили, что профилировщики добавляли столько накладных расходов к выполнению приложения, что данные о производительности были искажены. Эти профилировщики фактически способствовали возникновению проблемы, а не ее решению. Кроме того, ряд инструментов профилирования очень инвазивны для кода приложения, поскольку они могут фактически потребовать изменения кода, чтобы профилировщик мог проводить точные измерения. В результате таких действий разработчик может потерять контроль над своим кодом.

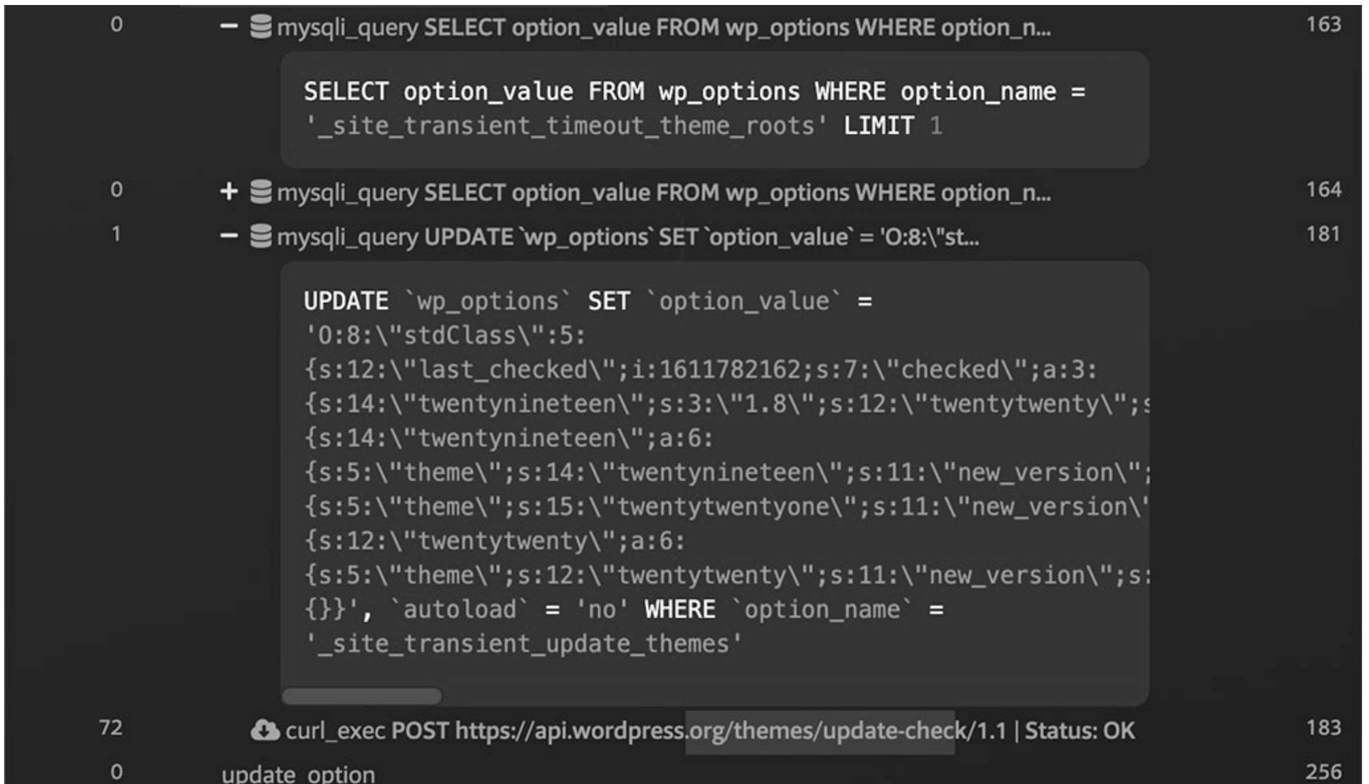

Рис. 1. Пример работы профилировщика Prefix [3]

Однако в последнее время стало появляться новое поколение инструментов профилирования. Современные профилировщики не подвержены многим ограничениям своих предшественников и действительно ускоряют процесс оптимизации.

На сегодняшний день выделяют два основных вида профилирования кода — на стороне сервера и на стороне рабочего стола.

Профилировщик на стороне сервера отслеживает эффективность ключевых методов в предпроизводственной или производственной среде. Он измеряет время транзакций, например, отслеживает, сколько длится веб-запрос, а также обеспечивает более прозрачную видимость ошибок и журналов. Примером серверного профилировщика может быть инструмент управления производительностью приложений.

Профилирование кода на стороне рабочего стола происходит медленнее и требует больших накладных расходов, что потенциально может привести к тому, что приложение будет работать намного медленнее, чем должно. Профилировщик такого типа обычно отслеживает производительность каждой строки кода в каждом отдельном методе. Эти типы профилировщиков также анализируют выделение памяти и позволяют установить какой метод использует больше всего ресурсов процессора.

В настоящее время активно разрабатывается и используется третий вид профилировщиков, который называется гибридным. Гибридные профилировщики кода объединяют ключевые данные из серверного профилирования с данными, полученными на стороне рабочего стола для ежедневного использования. Эти профилировщики предоставляют информацию на уровне сервера в сочетании с возможностью отслеживать ключевые методы, каждую транзакцию, зависимости, ошибки и журналы.

В качестве примера профилировщика на стороне сервера можно привести Retrace. В целом Retrace позволяет:

1. Собирать все фреймворки и зависимости автоматически.
2. Просматривать подробные снимков того, что делает код и сколько времени это занимает.
3. Отслеживать каждый SQL-запрос, выполняемый кодом.
4. Анализировать использование и производительность везде, где код отправляет HTTP-запросы.
5. Профилировать и понимать производительность асинхронного кода.

При работе с журналами Retrace дает возможность:
- объединять все журналы по всем приложениям и серверам;
- осуществлять просмотр и поиск по всем журналам приложения и сервера;
- переходить от записи журнала к полной трассировке транзакции;
- проводить более эффективный анализ с помощью тегов журнала и структурированного ведения журнала;
- настраивать и контролировать автоматические запросы журналов.

Кроме того, благодаря ему можно быстро определить, какая часть стека является узким местом, для этого проводится мониторинг удовлетворенности пользователей, отслеживаются развертывания, определяются медленные зависимости, оценивается производительность приложения.

Реальных решений для гибридного профилирования на сегодняшний день очень мало. Среди них можно отметить Prefix. Этот профилировщик позволяет обнаруживать неэффективные SQL-запросы, ORM-генерированные запросы и ранее неизвестные узкие места. Также он дает возможность отслеживать каждый параметр вызова SQL, затронутую запись и время загрузки, обнаружить шаблоны N + 1 (см. рис. 1).

Отдельный акцент необходимо сделать на том, что Prefix оснащен мощными средствами профилирования и трассировки кода Stackify, которые дают возможность выявить неэффективные зависимости. Это особенно полезно при работе с устаревшим кодом, разделами фреймворка и поиском скрытых исключений.

Отдельного внимания и высокой оценки заслуживает профилировщик AQtime, который представляет собой набор инструментов SmartBear для профилирования производительности, отладки памяти и ресурсов для компиляторов Microsoft, Borland, Java, Intel, Compaq и GNU. AQtime имеет очень широкий набор функций, в число которых входит:

1. Профилировщик производительности — этот профилировщик помогает находить плохо работающие функции и способствует их отладке. Он отслеживает все вызовы методов в приложении, подсчитывает их и следит за иерархией вызовов на текущей основе.
2. Профилировщик распределения — помогает определить, правильно ли приложение освобождает память. Это делается путем отслеживания использования памяти в 32-битных и 64-битных приложениях во время выполнения.
3. Профилировщик покрытия — позволяет узнать, какая часть кода на самом деле выполняется и тестируется. Он определяет, выполнялась ли процедура или строка во время профилировки и как каждый раз она выполнялась.
4. Профилировщик трассировки исключений подтверждает, что соответствующие сообщения об ошибках отображаются при определенных обстоятельствах. Он следит за выполнением приложения и (если необходимо) выводит информацию об исключительных случаях.
5. Профилировщик трассировки функций показывает, какой код вызывается и когда, позволяя убедиться, что используется наиболее эффективный путь кода. Он исследует маршрут и порядок, в котором процедуры вызываются во время выполнения приложения.
6. Загрузка библиотек Tracer Profiler. Многократная загрузка и выгрузка dll может значительно замедлить работу приложения, этот профилировщик определяет, какие библиотеки динамической компоновки были загружены и выгружены профилированным приложением и сколько раз они загружались и выгружались.
7. Профилировщик эмулятора сбоев — проверяет, содержит ли приложение код, который правильно обрабатывает сбои различных приложений.
8. Профилировщик соответствия платформы — позволяет определить, может ли профилируемое приложение работать в определенной операционной системе.
9. Ресурсный профайлер — проверяет, правильно ли приложение использует ресурсы Windows и соответствующим ли образом высвобождает эти ресурсы. Он следит за распределением и перераспределением ресурсов и обращается к процедурам управления ресурсами.
10. Профилировщик подсчета ссылок — отслеживает количество ссылок на объекты интерфейса в профилированном приложении.
11. Профилировщик ссылок на диаграмму последовательности — создает графическую карту того, как выполняются вызовы функций, что позволяет ускорить отладку кода. Он анализирует последовательность вызовов функций в приложении, а затем строит диаграмму вызовов функций в стиле UML. Это позволяет отслеживать связи между методами и функциями без запуска приложения.
12. Профилировщик статического анализа — идентифицирует экспозиции, которые могут быть написаны не для оптимальной производительности, анализируя отладочную информацию или метаданные. При этом анализируется большой объем информации, такой как количество циклов в рути-

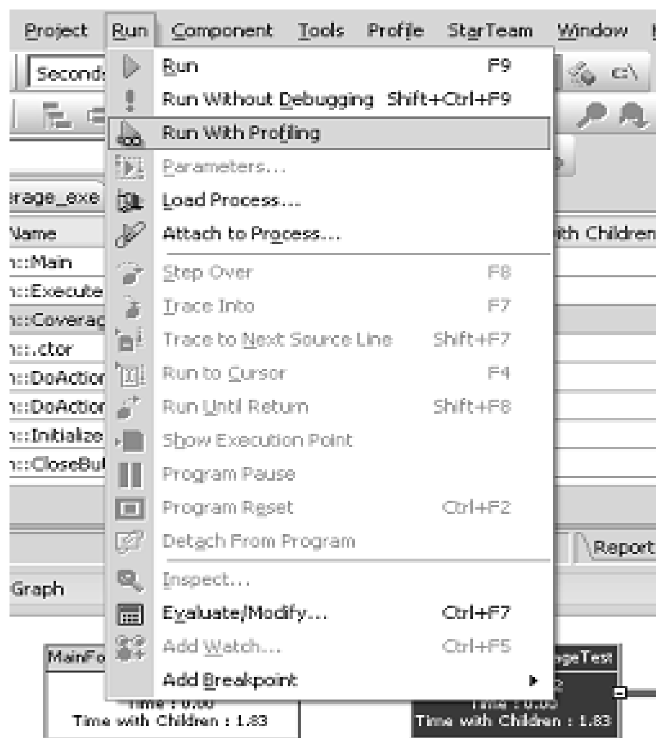

Рис. 2. Меню AQtime, интегрированное в Visual Studio

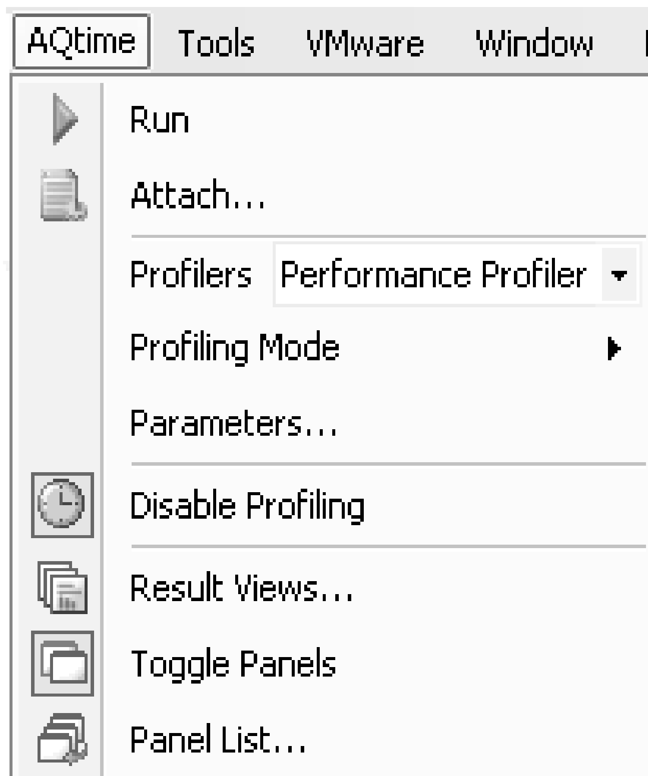

Рис. 3. Меню AQtime, интегрированное в Borland Developer Studio

не, размер рутины в байтах, все возможные ветви кода в приложении и многое другое.

13. Профилировщик SQL — измеряет производительность SQL-запросов или хранимых процедур SQL, вызываемых через Borland Database Engine (BDE).
14. Профилировщик неиспользуемых модулей VCL — помогает определить, какие модули VCL на самом деле не используются в приложении.

Несомненным преимуществом AQtime является его полная интеграция в Visual Studio и Borland Developer Studio, о чем наглядно свидетельствуют рис. 2 и 3 [4].

В специальной литературе помимо профайлеров кода — на стороне сервера и на стороне рабочего стола также выделяют два типа инструментальных профилировщиков: профилировщики, которые модифицируют исходный код и иерархические профилировщики.

Профилировщики, изменяющие исходный код, создают несколько проблем. Они имеют тенденцию конфликтовать с системами управления исходным кодом. Также они не всегда надежно анализируют источник, для которого предназначены. Фактически, поскольку добавление и удаление инструментария может быть достаточно сложным, профилировщики, изменяющие исходный код, часто предлагают пользователям работать с копией исходного кода проекта, чтобы избежать возможного повреждения. Кроме того, в лучшем случае эти профилировщики могут вставлять свой инструментальный код только в начале процедуры в исходный код. На этом этапе настройка процедуры уже запущена (для кадров стека, локальных переменных, параметров). В небольших процедурах настройка может занимать значительную часть времени выполнения. Тем не менее, невозможно рассчитать время самой установки с помощью профилировщика, изменяющего исходный код.

Иерархические профилировщики работают строго во время выполнения программы. Они встраивают свои инструменты непосредственно в исполняемый код приложения после его загрузки в память. Исходный код для этого не требуется, поэтому нет риска его испортить. Поскольку иерархический профилировщик работает заново при каждом выполнении, достаточно легко найти медленный код, затем попробовать улучшить исходный код, перекомпилировать и снова протестировать.

Оптимизация — это метод преобразования программы, который пытается улучшить код, заставляя его потреблять меньше ресурсов (т.е. ЦП, память) и обеспечивать высокую скорость [5]. В процессе оптимизации

общие программные конструкции высокого уровня заменяются более эффективными кодами низкоуровневого программирования. Процесс оптимизации кода должен следовать трем правилам:

1. Выходной код не должен изменять смысл программы.
2. Целью оптимизация является увеличение скорости работы программы и, если возможно, она должна потребовать меньше ресурсов.
3. Оптимизация должна быть быстрой и не задерживать общий процесс компиляции.

Попытки оптимизировать код могут быть предприняты на разных уровнях компиляции процесса. Вначале пользователи могут изменить / переставить код или использовать лучшие алгоритмы для написания кода. После генерации промежуточного кода компилятор может модифицировать промежуточный код, вычисляя адреса и улучшая циклы. При создании целевого машинного кода компилятор может использовать иерархию памяти и регистры ЦП.

Простым методам и механизмам оптимизации в современной литературе уже уделено достаточно много внимания, поэтому представляется целесообразным рассмотреть более подробно расширенные методы оптимизации.

1. Метод Hill Climbing или метод восхождения на холм — это эвристический поиск, используемый для задач математической оптимизации в области искусственного интеллекта. Учитывая большой набор входных данных и хорошую эвристическую функцию, он позволяет найти достаточно хорошее решение проблемы. Метод восхождения на холм решает проблемы, в которых необходимо максимизировать или минимизировать реальную функцию, выбирая значения из заданных входных данных. «Эвристический поиск» означает, что этот алгоритм может не найти оптимального решения проблемы. Однако в разумные сроки позволит получить наилучшее решение. Эвристическая функция является функцией, которая будет ранжировать все возможные варианты на любой стадии ветвления в алгоритме поиска на основе имеющейся информации. Это помогает алгоритму выбрать лучший маршрут из возможных.
2. Simulated Annealing — стохастический алгоритм глобальной поисковой оптимизации. Этот алгоритм использует случайность как часть процесса поиска, что делает его подходящим для нелинейных целевых функций, когда другие алгоритмы локального поиска работают плохо. Подобно алгоритму локального поиска стохастического восхождения на холм, Simulated Annealing изменяет единичное решение и ищет относительно локальную область пространства поиска, пока не будет обнаружен локальный оптимум. В отличие от алгоритма восхождения на холм, он может принимать худшие решения в качестве текущего рабочего решения. Вероятность принятия худших решений начинается с высокой точки в начале поиска и уменьшается с прогрессом поиска, давая алгоритму возможность сначала найти область для глобального оптимума, избегая локального оптимума, а затем подняться на высоту самого оптимума [6].
3. Генетический алгоритм (ГА) — метод локального поиска, используемый для обнаружения приблизительных решений задач оптимизации на случайной основе. Под случайной основой понимается, что для поиска решения с использованием ГА случайные изменения применяются к текущим решениям для генерации новых.

ГА имеет ряд преимуществ, которые делают его очень популярными и позволяют использовать в качестве метода оптимизации в процессе составления программного кода. Итак, ГА:

1. Не требует какой-либо производной информации (которая может быть недоступна для многих реальных проблем).
2. Работает быстрее и эффективнее по сравнению с традиционными методами.
3. Имеет очень хорошие параллельные возможности.
4. Оптимизирует как непрерывные, так и дискретные функции, а также многокритериальные задачи.
5. Предоставляет список «хороших» решений, а не одно решение.
6. Всегда получает ответ на проблему, который со временем становится более оптимальным.
7. Демонстрирует высокую эффективность, когда пространство поиска очень велико и задействовано большое количество параметров.

В тоже время необходимо отметить, что ГА имеет и ряд ограничений:

1. ГА не подходит для решения всех проблем, особенно для простых задач, по которым доступна производная информация.
2. Значение пригодности вычисляется повторно, что может потребовать больших вычислительных ресурсов для некоторых задач.
3. Поскольку решение является стохастическим, нет никаких гарантий оптимальности или качества решения.
4. Если ГА не реализован должным образом, он может не привести к оптимальному решению.

Таким образом, подводя итоги проведенного исследования, можно отметить следующее. За последние десятилетия стремительными темпами растет объем внедрения информационно-управляющих систем с интенсивным использованием программного обеспечения, что актуализирует задачу оптимизации процесса программирования через призму проверки безотказности кода, его эффективности и корректной работы приложения в целом. В процессе исследования рассмотрены различные методы профилирования, приведены примеры конкретных решений с детальным описанием их возможностей. Также отдельное внимание уделено современным, расширенным методам оптимизации, которые позволяют получить наилучшие решения с минимальными затратами и существенной отдачей.

ЛИТЕРАТУРА

1. Brais, Hadi Streamline Code with Native Profile-Guided Optimization // MSDN magazine. 2015. Volume 30: Number 9; pp 52–54.
2. López, Jorge; Kushik, Natalia; Yevtushenko, Nina Source code optimization using equivalent mutants // Information and software technology. 2018. Volume 103; pp 138–141.
3. Prefix URL: https://demo.prefix.io/
4. Memory and Performance Profiling Tool for Mission Critical Code URL: https://smartbear.com/product/aqtime-pro/overview/
5. Teixeira, Thiago SFX; Gropp, William; Padua, David Managing code transformations for better performance portability // The international journal of high performance computing applications. 2019. Volume 33: Number 6; pp 1290–1306.
6. Monmarché, Pierre Simulated annealing in Rd with slowly growing potentials // Stochastic processes and their applications. 2021. Volume 131; pp 276–291.

© Базарова Анна Максимовна (anna_sh94@inbox.ru).
Журнал «Современная наука: актуальные проблемы теории и практики»

Ухтинский государственный технический университет