DOI: 10.25586/RNU.V9187.21.01.P.159 УДК 004.051

К.Д. Новиков, М.В. Раскатова

# ОПТИМИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рассматриваются основные ошибки оптимизации программного обеспечения, способы ускорения работы приложений, а также освещаются спорные вопросы, затрагивающие написание программ. Представлены результаты исследований различных способов оптимизации программного кода. На основании полученных результатов исследования сделан вывод, какие правила следует соблюдать при создании исходного кода программы. Приведен сравнительный анализ компиляторов различных языков программирования и проводится исследование на тему оптимизации программного обеспечения с помощью компиляторов.
*Ключевые слова:* программирование, оптимизация, программное обеспечение, языки программирования, интернет.

K.D. Novikov, M.V. Raskatova

# SOFTWARE OPTIMIZATION

The article deals with the main mistakes in software optimization, talks about the ways to speed up applications, as well as covers controversial issues affecting program writing. The paper studies different ways to optimize the program code and provides the results of the research. Beside the above mentioned, based on the research results the conclusion is made which rules should be followed while creating a source code of a program. The paper also contains a comparative analysis of compilers for different programming languages and research on how these compilers deal with software optimization.
*Keywords:* programming, optimization, software, programming languages, Internet.

## *Введение*

История профессии программиста берет свое начало с 1843 г., когда был написан первый в истории документально зафиксированный программный код. В современном обществе технологии программирования шагнули далеко вперед, появилось множество вспомогательных инструментов, компиляторов, языков программирования, интерпретаторов и проч. Можно сказать, что написание программного кода сегодня – задача существенно облегченная по сравнению с тем временем. Наращивание новых вычислительных мощностей привело к тому, что в настоящее время от программистов не требуется написания оптимизированного исходного кода; главное требование к человеку, создающему программный код, – умение решать поставленную задачу, следуя определенным паттернам разработки, при этом нет большой разницы, будет ли задача выполняться условные 0,1 или 0,105 секунды.

Ослабление требований к оптимизации программного обеспечения привело к тому, что современные программисты, к сожалению, совершенно забывают об оптимизации [10]. К примеру, выбор изначально не оптимального алгоритма в сочетании с плохо написанными циклами и незнанием простых вещей вполне вероятно приведут к тому, что программа будет работать на порядок медленнее, чем могла бы, хотя поставленная зада-

ча будет выполняться. Не стоит забывать, что чем крупнее проект, тем большее влияние на производительность могут оказывать такие простые вещи, как правильно написанный цикл, особенно если он выполняется в программе бесчисленное множество раз [4, 7].

Все незначительные на первый взгляд детали, такие, например, как использование неподходящих типов переменных, выбор не самого лучшего алгоритма при возможности использовать более производительный, приводят современные приложения к состоянию, когда возникают проблемы с длительностью загрузки, торможением при работе программы.

Целью данной статьи является исследование влияния относительно простых принципов написания программного кода на производительность вычислительной системы в целом, а также выявление неэффективных методов оптимизации программного обеспечения.

С точки зрения обыкновенного программиста иногда не совсем понятно, какие принципы оптимизации исходного кода актуальны на текущий момент, несмотря на широкий спектр статей, написанных на эту тематику. Вышесказанное актуально в наше время в связи с непрерывным развитием технологий и языков программирования. Новые версии компиляторов позволяют писать более примитивный код за счет того, что оптимизация некоторых вещей выполняется без участия человека на этапе трансляции или компоновки исходного кода. Так как процесс совершенствования компиляторов непрерывен, возникают заблуждения относительно оптимизации программ. Иначе говоря, то, что было актуально во времена расцвета Pascal, уже не будет актуально на современных версиях NETCore, особенно при использовании последних версий IDE.

*Общая оптимизация*

Самым важным пунктом, значимость которого для быстродействия программного кода приложения сложно переоценить, является выбор оптимального алгоритма для решения поставленной задачи. Однако нельзя назвать какие-то определенные критерии правильности и оптимальности выбранного алгоритма – все упирается либо в уже известные способы решения поставленной задачи, либо в знания и интуицию программиста, хотя в некоторых случаях все-таки можно привести выбранный алгоритм под определенные математические критерии, что позволит однозначно сказать, является алгоритм оптимальным или нет [8].

В качестве примера, когда становится возможен анализ алгоритма, можно привести использование «жадных» алгоритмов (greedy algorithm), которые на каждом локальном шаге делают наилучший выбор, надеясь на то, что итог также выйдет наилучшим. Если программист решил использовать такой алгоритм, то существует способ проверить, будет ли greedy-алгоритм оптимальным. Эта проверка осуществляется с использованием матроидов.

Матроидом является пара $(X, I)$, в которой $X$ – это конечное множество, носитель матроида, а $I$ – это некоторое множество подмножеств $X$, называющееся семейством независимых множеств. При этом должны быть соблюдены следующие условия.

1. Множество $I$ не должно быть пустым. Исходное множество $X$ при этом может являться пустым, но $I$ должно включать в себя хотя бы один элемент (множество, содержащее пустое множество): $I = \{\{\emptyset\}\}$ или $\emptyset \in I$.

2. Для любого подмножества $A$ элементов из $I$ должно выполняться следующее условие: $A \subset B, B \in I \rightarrow A \in I$.

3. Если существуют такие множества $A$ и $B$, принадлежащие множеству $I$, что $|A| < |B|$, то существует такой элемент $x \in B, x \notin A$, что $\{x\} \cup A \in I$.

Обобщая, если программист использует «жадный» алгоритм и в состоянии доказать, что объект его алгоритма является матроидом, то согласно теореме Радо – Эдмондса алгоритм будет корректным и оптимальным.

Существует еще один известный способ оптимизации выбранного алгоритма – динамическое программирование. Оно подразумевает под собой решение сложной задачи разбиением на более мелкие подзадачи, зачастую являющиеся тривиальными. При этом каждая из мелких подзадач должна решаться в общей системе не более одного раза. Иначе говоря, необходимо запоминать результаты вычисления функций с определенным набором аргументов, чтобы избежать повторного выполнения одного и того же участка кода.

Возможность запоминать результаты функции встроена в некоторые языки программирования (например, Perl, Common Lisp). Однако в большинстве популярных языков, таких как C++, такая возможность изначально отсутствует, и требуется установка дополнительных расширений [4, 7].

Следующей ошибкой, часто встречающейся в исходных кодах приложений, является неуместное использование конструкций ветвления. Рассмотрим пример кода на языке C++ с использованием последней актуальной версии Visual Studio (на момент написания статьи):

```
classTest {
public:
void PrintFirst(std::strings) {
if (s!="test") {
          std::cout <<".";
     }
  }
           };
int main() {
Test *t = newTest();
  std::cin >> s;
unsignedint start_time = clock();
for (int i = 0; i < 100000; i++) {
if (s !="test") {
        t->PrintFirst(s);
     }
  }
unsignedint end_time = clock();
unsignedint search_time = end_time - start_time;
  std::cout <<"\n"<<"Algorithm time: "<< search_time <<" ms";
        }
```

Как можно заметить, в код намеренно вставлена лишняя конструкция ветвления внутри метода класса. Точно такая же проверка находится в основной части программного кода. С логической точки зрения, так как обе конструкции находятся внутри цикла, каждый шаг цикла будут выполняться две одинаковые проверки. Проведем проверку для

этого кода, а также для случая, когда избыточная конструкция ветвления убрана из исходного кода; помимо этого протестируем аналог данного кода на языке Python. Результаты представлены в таблице 1.

*Таблица 1*

**Усредненное время выполнения ветвлений**

| Язык программирования | С удвоенным IF | Без удвоенного IF |
|---|---|---|
| C++ | 439 мс | 431 мс |
| Python | 3170 мс | 3234 мс |
| C# | 820 мс | 718 мс |

Из результатов, представленных в таблице, видно, что корреляция времени исполнения кода и избыточных проверок явно присутствует только на платформе .NET. Это может говорить о том, что современный компилятор языка C++ способен оптимизировать подобные участки кода. Однако полученные результаты совсем не говорят о том, что компилятор языка C# плохо выполняет свои функции, – он работает с использованием других методов.

Рассмотрим пример, в котором явно будут прослеживаться возможности оптимизации кода компилятором .NETCore, – сравнение двух байт с использованием условного оператора и без него:

```
        publicint CompareBytes(byte a, byte b)
    {
unchecked
        {
int ab = (int)a - (int)b;
int ba = (int)b - (int)a;
return (ab >> 31) | (int)((uint)ba >> 31);
         }
    }
publicint CompareIF(byte a, byte b)
     {
if (a < b)
        {
return -1;
         }
elseif (a > b)
{
return 1;
          }
elsereturn 0;
              }
```

Казалось бы, очевидно, что с практической точки зрения метод без использования условного оператора должен быть на порядок быстрее его аналога, однако тестирование показывает совершенно другие усредненные значения: 60 мс у первого метода в сравнении с 7 мс у второго метода. Тем не менее подход, подобный первому методу, в определен-

ных местах способен сократить время выполнения программы. Очевидно, что подобные места не должны оптимизироваться компилятором.

*Программная оптимизация*

Использование неподходящих типов данных может повлечь за собой несущественное снижение в скорости выполнения программы, однако если неправильный тип переменной выбран внутри цикла с огромным числом итераций, то деградация быстродействия может стать существенной.

Самым простым примером может служить использование List в методах, от которых не требуется изменение переменной, а нужно лишь отображение. Для простого отображения данных лучше использовать преобразование to Array. Это утверждение основано на том, что класс List практически всегда является «оберткой» для обычного массива данных, и его использование неминуемо приведет к выполнению дополнительных инструкций процессором.

Помимо вышесказанного, часто можно слышать утверждение, что не стоит использовать циклы foreach, если существует возможность использования цикла for. Это правдивое утверждение, но существует исключение – работа с массивами. Проведем тестирование (табл. 2).

*Таблица 2*

**Усредненные результаты времени работы программы с использованием Array, List внутри циклов for, foreach, при $N = 10000$**

| Вид преобразования | For | Foreach |
|---|---|---|
| List | 102 мкс | 154 мкс |
| Array | 59 мкс | 42 мкс |

Быстродействие цикла foreach в данном случае связано с тем, что компилятор выполняет максимально оптимизированное итерирование массива по индексу без каких-либо проверок. Для случаев, когда в коде используется не массив, предпочтительно использование циклов for и while. Вышесказанное актуально для .NETCore, на котором проводилось тестирование.

При написании оптимизированного программного кода существует еще один, не самый очевидный, нюанс – быстродействие жесткого диска. Если говорить обобщенно, то при постоянной работе с небольшими файлами на винчестере имеет смысл сохранить поток байт в оперативную память и производить работу непосредственно с ним, чем постоянно считывать и записывать изменения на диск, так как очевидно, что скорость работы оперативной памяти многократно превосходит скорость считывания/записи жестких дисков.

Верным будет утверждение, что никто лучше компилятора не сможет оптимизировать ваш программный код. Следовательно, чем более конкретно программист описывает код, тем сильнее он сможет оптимизировать его компилятор (помимо оптимизации такой код станет более читабельным). В качестве примера можно привести добавление ключевого слова readonly в те места кода, где это оправдано.

Если рассматривать более тонкие методы оптимизации, то можно рассмотреть постфиксные и префиксные операторы. Как известно, постфиксные операторы медленнее

префиксных, однако это неверно для примитивных типов данных на современных компиляторах: произойдет компиляция в идентичный код. Но если рассматривать объекты, то здесь уже не все так однозначно: если компилятор «не додумается» оптимизировать код, последует деградация быстродействия. Причина – операторы постфикса обязаны вернуть неизменную версию значения независимо от того, нужен результат программисту или нет, а значит, будет создана копия.

*Заключение*

При рассмотрении вопросов оптимизации программного обеспечения можно сделать однозначный вывод: на быстродействие существенно могут повлиять три составляющие – ошибки с алгоритмической точки зрения, неподходящий выбор языка программирования и отсутствие узкой оптимизации программного кода.

На подготовительном этапе создания программного обеспечения необходимо учитывать специфику решаемой задачи и решать, насколько критична будет производительность приложения. Если задача максимально мала, разумно будет решать задачу на самом простом и удобном для программиста языке программирования, например на Python. Если же стоит задача обработки огромных данных массивов, то имеет смысл использовать максимально низкоуровневый язык программирования, такой как C или C++.

Таким образом, для достижения максимальной производительности необходимо тестировать каждый отдельно взятый кусок программного кода, так как общепринятые нормы иногда могут негативно отразиться на быстродействии приложения.

## Литература

1. *Гантерот К.* Оптимизация программ на C++. Проверенные методы повышения производительности. 1-е изд. М.: Вильямс, 2017. 400 с.
*2. Голдштейн С., Зурбалев Д., Флатов И.* Оптимизация приложений на платформе .Net. 2-е изд. М.: ДМК Пресс, 2014. 524 с.
*3. Джепикс Ф., Троелсен Э.* Язык программирования C# 7 и платформы .NET и .NET Core. 8-е изд. М.: Вильямс, 2018. 1328 с.
*4. Липпман С.Б., Лажойе Ж., Му Б.Э.* Язык программирования C++. Базовый курс. 5-е изд. М.: Вильямс, 2017. 1118 с.
*5. Лутц М.* Изучаем Python. 4-е изд. М.: Символ-Плюс, 2011. 1280 с.
*6. Макконнелл С.* Совершенный код. Мастер-класс. М.: БХВ-Петербург, 2017. 896 с.
*7. Мейерс С.* Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14. М.: Вильямс, 2018. 304 с.
*8. Ньюланд К., Эванс Б.* Java. Оптимизация программ. Практические методы повышения производительности приложений в JVM. М.: Вильямс, 2019. 448 с.
*9. Рамальо Л.* Python. К вершинам мастерства. М.: ДМК Пресс, 2016. 768 с.
10. Ускорение кода на Python средствами самого языка / Хабр [Электронный ресурс]. – URL: https://habr.com/ru/post/124388/ (дата обращения 11.11.2020).

## Literatura

1. *Ganterot K.* Optimizaciya programm na C++. Proverennye metody povysheniya proizvoditel'nosti. 1-e izd. M.: Vil'yams, 2017. 400 s.

2. *Goldshtejn S., Zurbalev D., Flatov I.* Optimizaciya prilozhenij na platforme .Net. 2-e izd. M.: DMK Press, 2014. 524 s.
3. *Dzhepiks F., Troelsen E.* Yazyk programmirovaniya C# 7 i platformy .NET i .NET Core. 8-e izd. M.: Vil'yams, 2018. 1328 s.
4. *Lippman S.B., Lazhoje Zh., Mu B.E.* Yazyk programmirovaniya C++. Bazovyj kurs. 5-e izd. M.: Vil'yams, 2017. 1118 s.
5. *Lutc M.* Izuchaem Python. 4-e izd. M.: Simvol-Plyus, 2011. 1280 s.
6. *Makkonnell S.* Sovershennyj kod. Master-klass. M.: BKhV-Peterburg, 2017. 896 s.
7. *Mejers S.* Effektivnyj i sovremennyj S++. 42 rekomendacii po ispol'zovaniyu C++11 i C++14. M.: Vil'yams, 2018. 304 s.
8. *N'yuland K., Evans B.* Java. Optimizaciya programm. Prakticheskie metody povysheniya proizvoditel'nosti prilozhenij v JVM. M.: Vil'yams, 2019. 448 s.
9. *Ramal'o L.* Python. K vershinam masterstva. M.: DMK Press, 2016. 768 s.
10. Uskorenie koda na Python sredstvami samogo yazyka / Khabr [Elektronnyj resurs]. – URL: https://habr.com/ru/post/124388/ (data obrashcheniya 11.11.2020).

DOI: 10.25586/RNU.V9187.21.01.P.165 УДК 004.4

С.В. Бондаренко, Т.В. Жукова, Э.М. Вихтенко

# РАЗРАБОТКА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО КОМПОНЕНТНОГО ФРЕЙМВОРКА, РЕАЛИЗУЮЩЕГО ПАРАДИГМУ MVC

Разработан фреймворк на языке PHP, поддерживающий парадигму разделения данных приложения. Реализована возможность подключения к нескольким видам баз данных благодаря использованию PDO (PHP Data Objects). Описаны реакции приложения на действия пользователя. Разработанное программное средство использует для работы с базами данных классы ActiveRecord, выполняет перехват и обработку ошибок, обеспечивает ввод и валидацию форм, аутентификацию и авторизацию пользователей, позволяет генерировать исходный PHP-код для CRUD-операций, а также поддерживает шаблоны оформления веб-страниц.
*Ключевые слова:* MVC-фреймворк, язык PHP, веб-сайт, CRUD-операции, генерация кода, виджет.

S.V. Bondarenko, T.V. Zhukova, E.M. Vikhtenko

# DEVELOPMENT OF OBJECT-ORIENTED COMPONENT FRAME IMPLEMENTING THE MVC PARADIGM

A framework in PHP has been developed that supports separation of data paradigm. Implemented the ability to connect to several types of databases using PDO (PHP Data Objects). The application's reactions to user actions are described. The developed software tool uses the ActiveRecord classes to work with databases, intercepts and handles errors, provides form input and validation, user authentication and authorization, allows generating PHP source code for CRUD operations, and also supports web page design templates.
*Keywords:* MVC framework, PHP language, website, CRUD operations, code generation, widget.