Математические структуры и моделирование
2015. № 1(33). С. 83–92

УДК 004.051

ОПТИМИЗАЦИЯ ВРЕМЕНИ РАБОТЫ АЛГОРИТМОВ ПРИ РАЗРАБОТКЕ ПЛАТФОРМЫ АЛГОРИТМИЧЕСКОЙ ТОРГОВЛИ ВАЛЮТАМИ

А.И. Журавлев
аспирант, e-mail: grusalex@gmail.com
Д.Н. Лавров
к.т.н., доцент, e-mail: lavrov@omsu.ru

Омский государственный университет им. Ф.М. Достоевского

Аннотация. В работе представлены подходы к уменьшению времени исполнения участков программного кода для алгоритмов, обладающих определённой сложностью. Приведённые примеры кода на языке C++ иллюстрируют данные подходы. Анализ примеров объясняет меньшее время их исполнения улучшенным использованием кэшей инструкций и данных. Описанные оптимизации применены при разработке банковской платформы алгоритмической торговли.

Ключевые слова: платформа алгоритмической торговли, программный код, время выполнения, кэши инструкций и данных.

Введение

Одной из главных характеристик любой программы является время, которое необходимо на выполнение её основных функций. Разные классы приложений имеют разные требования к данной характеристике. В частности эти требования очень строги применительно к платформам алгоритмической торговли различными финансовыми инструментами, например, ценными бумагами или валютами. Далее будут описаны и рассмотрены некоторые методы и решения, сокращающие временные затраты и применённые при разработке одной из таких систем, работающей на рынке обмена валют. Данная торговая платформа используется в крупном инвестиционном банке и характеризуется как система с ультра-низкой задержкой (ultra-low latency).

Особенности этого рынка — быстрая смена курсов обмена, происходящая за малые доли секунды, и высокая конкуренция на рынке, вызванная большим числом участников. При запоздалой реакции системы на изменения рынка её клиенты обслуживаются по неоптимальным ценам, что приводит к снижению конкурентоспособности приложения. Следовательно, разработчиками должна быть обеспечена высокая производительность данной торговой платформы. Более того, недостаточно просто реализовать требуемую функциональность, и даже недостаточно реализовать её с заданными временными характеристиками выполнения — в связи с высокой конкуренцией со стороны других торговых

платформ, необходимо постоянно и итеративно оптимизировать уже используемые решения.

1. Этапы оптимизационного процесса

Чтобы достичь высокой производительности программы, следует использовать быстрые алгоритмы — другими словами, алгоритмы с наименьшей сложностью из имеющихся. Применительно к банковской торговой платформе, выбранный, реализованный и протестированный алгоритм выпускается для работы в реальных условиях рынка. Однако в ходе анализа его использования может быть выявлено, что конкурирующие приложения, использующие тот же алгоритм, добиваются более значительных результатов за счёт меньшего времени работы. Таким образом, возникает необходимость оптимизировать программную реализацию используемого алгоритма.

Первым шагом данной оптимизации является просмотр (ревью) кода алгоритма для уточнения деталей реализации. При этом достаточно быстро могут быть найдены действия и операции, которые очевидно избыточны и должны быть устранены — например, распечатка в лог вспомогательных сообщений, оставшаяся после этапа отладки приложения и ненужная при работе в реальных условиях.

Затем проводится анализ того, возможно ли уменьшение времени выполнения алгоритма путём преобразования написанного ранее кода без изменения его математической модели, обладающей определённой сложностью. Далее приведены примеры такого анализа, в котором особое внимание зачастую уделяется использованию кэша инструкций и кэша данных. Разумеется, эти примеры не покрывают все множество возможных оптимизаций.

Прежде чем перейти к данным примерам, стоит упомянуть то, как оценивается время работы алгоритма, чтобы отслеживать влияние оптимизационных изменений. Обычно оптимизации подвергаются достаточно локальные части кода, например, отдельные методы или процедуры, на которых предполагается достижение большей производительности. Такие части выявляются путём анализа всего оптимизируемого сценария выполнения. Затем для них пишутся минимизированные тесты, что избавляет от необходимости запускать весь сценарий и позволяет фокусироваться лишь на его неэффективных участках. Если тесты воспроизводят недостаточную производительность, то покрываемый ими код, очевидно, стоит оптимизировать. Однако они могут продемонстрировать небольшое время исполнения кода, что может быть связано с внешними условиями относительно оптимизируемого и тестируемого кода. Например, если в сценарии выполнения происходит конвертация сообщения из одного формата в другой, то тест может свидетельствовать, что она занимает мало времени. Такая оценка, выдаваемая тестом, получается путём усреднения времени, затрачиваемого на многократную последовательную конвертацию. При этом стоит учитывать, что после выполнения первых тестовых итераций происходит заполнение кэшей, что и сокращает длительность последующих. С другой стороны, конвертация может происходить в реальном сценарии выполнения лишь

однократно, и влияние кэшей на неё отсутствует.

Таким образом, измерения времени не должны искажать реальное положение вещей.

Обозначенная выше торговая платформа разработана на языке C++, и для измерения времени работы кода может быть использована следующая функция, выполнение которой оценивается в 7 наносекунд. Это приемлемая точность для оценок в системах с ультра-низкой задержкой, так как, по определению, время реакции таких систем составляет менее 20 микросекунд [1] [2].

```
uint64_t rdtsc() {
    uint32_t a,d;
    asm volatile("rdtsc" : "=a" (a), "=d" (d));
    return ((uint64_t)a) | (((uint64_t)d)<<32;
}
```

Приведённая функция основана на применении ассемблерной инструкции rdtsc [3].

2. Удаление или перемещение лишнего кода

В следующем фрагменте кода вычисляется сумма произведений пар вещественных чисел, расположенных по соседству в векторе. Выполнение этого кода занимает 28 миллисекунд.

```
std::vector<double> values;
double result = 0.0;
int N = 10000000;
for (size_t i = 1; i < N; ++i) {
    result += values[i-1]*values[i];
}
```

Эта достаточно простая реализация может выступить в качестве прототипа, и специалисты в финансовых вычислениях могут потребовать изменить её с учётом бизнес-рисков. Например, по их утверждениям, вектор может содержать неинициализированные элементы или значения NaN (not-a-number), что может вызвать некорректную цену для клиентской сделки. В таком случае следует добавить контроль корректности вычислений, что достигается, в частности, путём самостоятельной реализации класса Double с переопределённым оператором умножения.

```
Double operator * (const Double & d) const {
    Double result;
    if (!m_empty && !d.m_empty) {
        result.m_value = m_value * d.m_value;
        result.m_empty = false;
    } else {
```

```
        print_error();
    }
    return result;
}

void Double::print_error() const {
    std::stringstream sstrm;
    sstrm << __FILE__ << ":" << __LINE__
    << "Uninitialized value!" << std::endl;
    std::cout << sstrm.str() << std::endl;
}
```

Как видно из кода, к обычному вещественному числу добавлен булевый флаг empty, позволяющий проверять, корректно ли инициализировано число. В случае ошибки производится её логирование, и производительностью этой операции можно пренебречь. Выполнение изменённого кода занимает 30 миллисекунд, что на 2 миллисекунды больше изначального результата. В данном случае дополнительные накладные расходы вполне приемлемы, так как они невелики и вызваны необходимостью снизить бизнес-риски при функционировании приложения.

Однако возможен и другой вариант реализации, без создания отдельной функции `print_error`. В таком варианте её код просто помещается в альтернативную часть ветвления. Многие программисты нередко практикуют такой подход, особенно если логика функции используется лишь в одном месте. Кроме того, на выделение и написание подобных отдельных методов также может требоваться дополнительное время, ценное при очень сжатых сроках разработки.

```
Double operator * (const Double & d) const {
    Double result;
    if (!m_empty && !d.m_empty) {
        result.m_value = m_value * d.m_value;
        result.m_empty = false;
    } else {
        std::stringstream sstrm;
        sstrm << __FILE__ << ":" << __LINE__
        << "Uninitialized value!" << std::endl;
        std::cout << sstrm.str() << std::endl;
    }
    return result;
}
```

Данный код выполняется за 40 миллисекунд, и накладные расходы составляют 12 миллисекунд, что в 6 раз больше по сравнению с предыдущим вариантом. Ненужный код, скрытый в функции `print_error`, не приводит к

заметному замедлению алгоритма в отличие от кода, написанного явно. Стоит отметить, что вызов данной функции может производиться при отладке и тестировании системы более часто, нежели в реальных условиях, так как некорректные данные, скорее всего, возникают лишь из-за дефектов других компонент системы, тестирование которых также не завершено.

3. Проверка необходимости исполнения кода

В реальной жизни люди редко ходят в гости к своим друзьям, не уточнив, дома ли они. Аналогичные ситуации программисты могут увидеть и при написании кода. В частности, это происходит при вызове функции, которая, возможно, ничего не делает и фактические работает вхолостую. Например, проверка на ошибки набора программных модулей может быть реализована следующим образом:

```
for (const auto & module: modules) {
    module.check_errors();
}
```

Код функции `check_errors` выглядит так:

```
void Module::check_errors() const {
    if (m_errors == 0) {
        return;
    }
    // big business logic
    std::stringstream sstrm;
    sstrm << __FILE__ << ":" << __LINE__ ;
    sstrm << " Our module has " << m_errors
    << " errors" << std::endl;
    std::cerr << sstrm.str() << std::endl;
}
```

Выполнение цикла обхода и проверки модулей занимает 29 миллисекунд. Однако приведённая ниже альтернативная реализация цикла, содержащая «лишнюю» проверку — inline-фунцию `has_errors`, работает за 6 миллисекунд (`check_errors` не имеет изменений).

```
for (const auto & module: modules) {
    if (module.has_errors()) {
        module.check_errors();
    }
}
```

4. Организация структур обрабатываемых данных

Многие платформы алгоритмической торговли валютами оперируют книгами заказов (order book) — структурами данных, содержащих курсы валют и их объёмы в совершаемых сделках. Обычно разные программисты реализуют книги заказов по-разному, исходя из собственного опыта и своих предпочтений.

Например, такая структура данных характерна для программирования в стиле языка C (size хранит число заполненных записей):

```
#define MAX_LEVEL 50
struct Book1 {
    size_t size;
    int prices[MAX_LEVEL];
    int volumes[MAX_LEVEL];
};
```

С другой стороны, нередка реализация, выглядящая в стиле языка Java:

```
struct Book2 {
    struct Level {
        int price;
        int volume;
    };
    size_t size;
    Level levels[MAX_LEVEL];
};
```

Тогда в приложении возможно написание двух вариантов кода, связанного с чтением книг заказов и записью в них. Допустим, что N=10000 и size=1. Запись:

A. Время исполнения 115 микросекунд.

```
for (size_t i = 0; i < N; ++i) {
    auto & book = books1[i];
    for (size_t j = 0; j < size; ++j) {
        book.prices[j] = ++v;
        book.volumes[j] = ++v;
    }
}
```

B. Время исполнения 43 микросекунды.

```
for (size_t i = 0; i < N; ++i) {
    auto & book = books2[i];
    for (size_t j = 0; j < size; ++j) {
        auto & level = book.levels[j];
        level.price = ++v;
        level.volume = ++v;
    }
}
```

Чтение: Время исполнения 50 микросекунд.

```
for (size_t i = 0; i < N; ++i) {
    const auto & book = books1[i];
    for (size_t j = 0; j < size; ++j) {
        sum += book.prices[j];
        sum += book.volumes[j];
    }
}
```

B. Время исполнения 23 микросекунды.

```
for (size_t i = 0; i < N; ++i) {
    const auto & book = books2[i];
    for (size_t j = 0; j < size; ++j) {
        const auto & level = book.levels[j];
        sum += level.price;
        sum += level.volume;
    }
}
```

5. Профилирование приведённых примеров

Чтобы исследовать и объяснить результаты, полученные в приведённых выше примерах, был использован компонент профилирования Callgrind, входящий в набор инструментов Valgrind — бесплатное средство динамического двоичного анализа программ. В частности, Callgrind позволяет проанализировать использование кэшей инструкций и данных.

Применительно к первому примеру, Callgrind сгенерировал данные, приведённые ниже (рис. 1), где красный цвет соответствует более медленной версии кода и используемые сокращения обозначают следующие характеристики [4]:

Ir — число чтения инструкций

Dr — число чтения данных

Dw — число записи данных

I1mr — число промахов при чтении для кэша инструкций первого уровня

D1mr — число промахов при чтении для кэша данных первого уровня

D1mw — число промахов при записи для кэша данных первого уровня

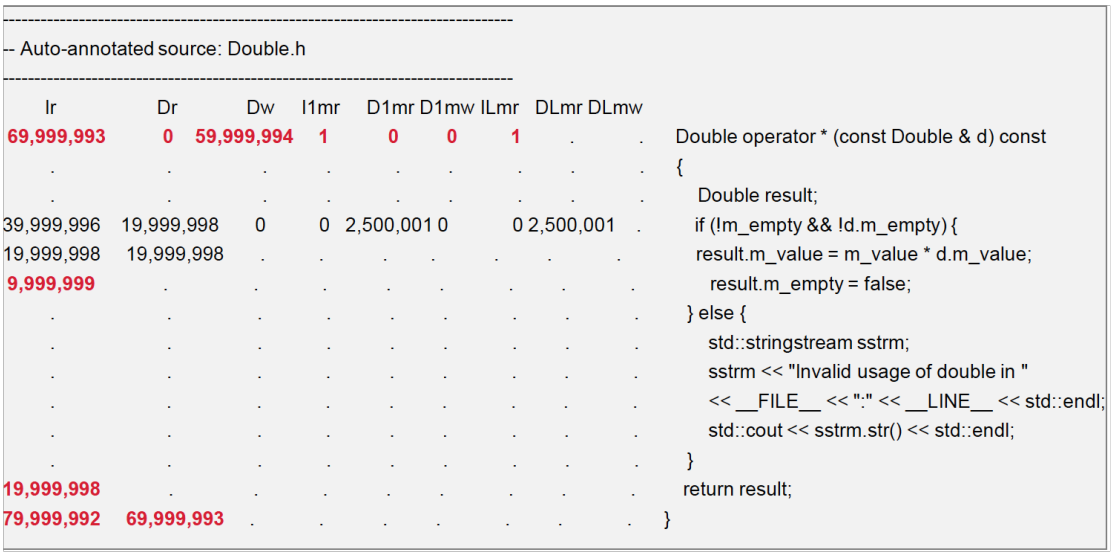
ILmr — число промахов при чтении для кэша инструкций последнего уровня

DLmr — число промахов при чтении для кэша данных последнего уровня

DLmw — число промахов при записи для кэша данных последнего уровня

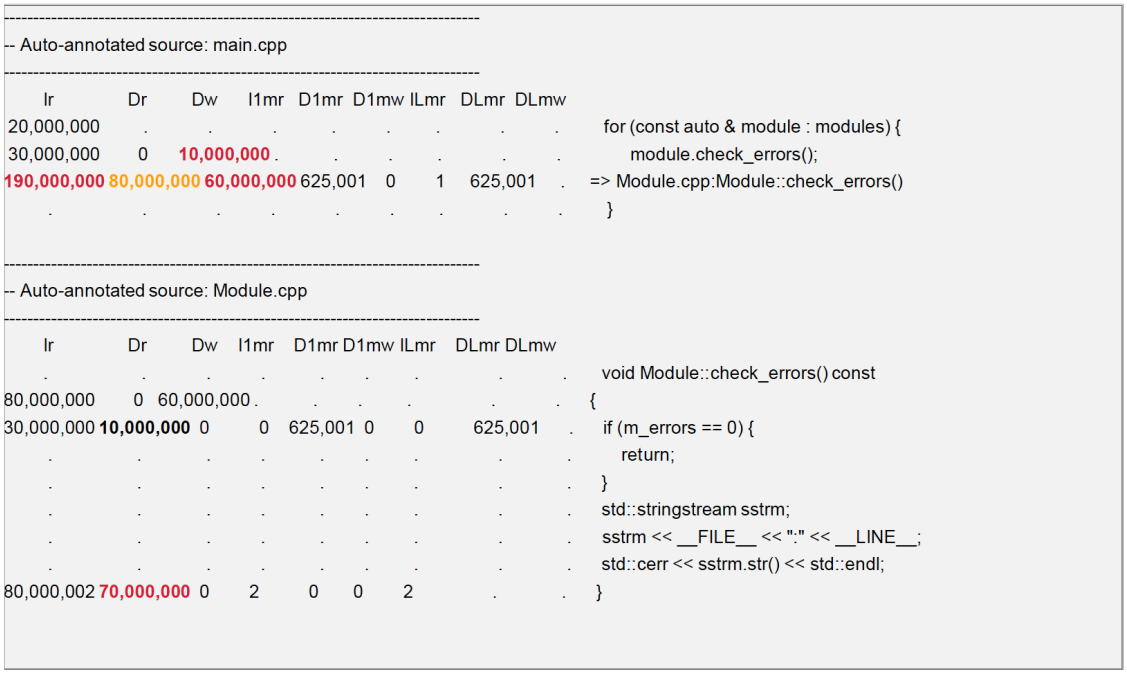
Замедление вызвано тем, что функция перестала быть встраиваемой (inline) и в поток выполнения явно добавился код всей функции. Компилятор не может определить, что добавленный код выполняется не часто, и всегда загружает его. К аналогичным заключениям приводит анализ второго примера (рис. 2).

Третий пример (случай записи) показывает, что более близкое расположение элементов level друг к другу способствует более оптимальному использованию



```
--------------------------------------------------------------------------------
-- Auto-annotated source: Double.h
--------------------------------------------------------------------------------
      Ir          Dr          Dw       I1mr     D1mr D1mw ILmr   DLmr DLmw
 69,999,993       0  59,999,994       1        0    0    1      .    .     Double operator * (const Double & d) const
       .          .           .       .        .    .    .      .    .     {
       .          .           .       .        .    .    .      .    .       Double result;
39,999,996 19,999,998          0       0 2,500,001 0         0 2,500,001  .     if (!m_empty && !d.m_empty) {
19,999,998 19,999,998          .       .        .    .    .      .    .       result.m_value = m_value * d.m_value;
 9,999,999          .          .       .        .    .    .      .    .         result.m_empty = false;
       .          .          .       .        .    .    .      .    .       } else {
       .          .          .       .        .    .    .      .    .         std::stringstream sstrm;
       .          .          .       .        .    .    .      .    .         sstrm << "Invalid usage of double in "
       .          .          .       .        .    .    .      .    .         << __FILE__ << ":" << __LINE__ << std::endl;
       .          .          .       .        .    .    .      .    .         std::cout << sstrm.str() << std::endl;
       .          .          .       .        .    .    .      .    .       }
19,999,998          .          .       .        .    .    .      .    .     return result;
79,999,992 69,999,993          .       .        .    .    .      .    .   }
```

Рис. 1. Удаление или перемещение лишнего кода



```
--------------------------------------------------------------------------------
-- Auto-annotated source: main.cpp
--------------------------------------------------------------------------------
      Ir          Dr          Dw       I1mr     D1mr D1mw ILmr   DLmr DLmw
 20,000,000       .           .       .        .    .    .      .    .     for (const auto & module : modules) {
 30,000,000       0  10,000,000       .        .    .    .      .    .       module.check_errors();
190,000,000 80,000,000 60,000,000 625,001     0    1    625,001  .   => Module.cpp:Module::check_errors()
       .          .           .       .        .    .    .      .    .     }

--------------------------------------------------------------------------------
-- Auto-annotated source: Module.cpp
--------------------------------------------------------------------------------
      Ir          Dr          Dw       I1mr     D1mr D1mw ILmr   DLmr DLmw
       .          .           .       .        .    .    .      .    .     void Module::check_errors() const
80,000,000         0  60,000,000       .        .    .    .      .    .   {
30,000,000 10,000,000          0       0  625,001 0         0  625,001  .     if (m_errors == 0) {
       .          .           .       .        .    .    .      .    .       return;
       .          .           .       .        .    .    .      .    .     }
       .          .           .       .        .    .    .      .    .     std::stringstream sstrm;
       .          .           .       .        .    .    .      .    .     sstrm << __FILE__ << ":" << __LINE__;
       .          .           .       .        .    .    .      .    .     std::cerr << sstrm.str() << std::endl;
80,000,002 70,000,000          0       2        0    0    2      .    .   }
```

Рис. 2. Проверка необходимости исполнения кода

кэша данных первого уровня в оптимизируемом цикле записи (рис. 3). Аналогичный результат получен и для цикла чтений.

```
--------------------------------------------------------------------------------
-- Auto-annotated source: main.cpp
--------------------------------------------------------------------------------
    Ir     Dr     Dw I1mr D1mr D1mw ILmr DLmr DLmw
20,000      .      .    .    .    .    .    .    .  for (size_t i = 0; i < N; ++i) {
     .      .      .    .    .    .    .    .    .    auto & book = books2[i];
     .      .      .    .    .    .    .    .    .    for (size_t j = 0; j < size; ++j) {
     .      .      .    .    .    .    .    .    .      auto & level = book.levels[j];
     .      .      .    .    .    .    .    .    .      level.price = ++v;
40,000  0  20,000  0    0   10,000   .    .    .      level.volume = ++v;
     .      .      .    .    .    .    .    .    .    }
     .      .      .    .    .    .    .    .    .  }

20,000      .      .    .    .    .    .    .    .  for (size_t i = 0; i < N; ++i) {
     .      .      .    .    .    .    .    .    .    auto & book = books1[i];
     .      .      .    .    .    .    .    .    .    for (size_t j = 0; j < size; ++j) {
     .      .      .    .    .    .    .    .    .      book.prices[j] = ++v;
40,000  0  20,000  0    0   20,000   .    .    .      book.volumes[j] = ++v;
     .      .      .    .    .    .    .    .    .    }
     .      .      .    .    .    .    .    .    .  }
```

Рис. 3. Организация структур обрабатываемых данных

6. Заключение

Приведённые примеры и их анализ демонстрируют, что уменьшение времени исполнения алгоритма возможно за счёт выбора более оптимальной программной реализации, приводящей к более рациональному использованию кэша инструкций и расположению данных. Таким образом, описанная методика совместного применения измерения времени исполнения и профилирования кода, учитывающая влияние расположения кода в потоке выполнения и организации структур данных, способна приводить к заметному сокращению времени работы отдельных участков алгоритмов.

Литература

1. NYSE Technologies Launches Market Data Platform V5. URL: `http://nyse.com/press/1245751792887.html` (дата обращения 15.12.2014).
2. NOne Trading Firm's View of Low Latency. URL: `http://intelligenttradingtechnology.com/blog/one-trading-firms-view-low-latency` (дата обращения 15.12.2014).
3. Using the RDTSC Instruction for Performance Monitoring. URL: `https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf` (дата обращения 15.12.2014).
4. Callgrind: a call-graph generating cache and branch prediction profiler. URL: `http://valgrind.org/docs/manual/cl-manual.html` (дата обращения 15.12.2014).

PERFORMANCE OPTIMIZATION OF THE ALGORITHMS USED IN DEVELOPING A PLATFORM FOR ALGORITHMIC CURRENCY TRADING

A.I. Zhuravlev
Post-graduate student, e-mail: grusalex@gmail.com
D.N. Lavrov
Ph.D. (Eng.), Associate Professor, e-mail: lavrov@omsu.ru

Omsk State University n.a. F.M. Dostoevskiy

Abstract. The article represents approaches to reducing of execution time for algorithms with particular complexity. Given examples of C++ code illustrate those approaches. It is shown that the execution time is reduced by more efficient use of instruction and data caches. Described optimizations are applied to improve a bank platform for algorithmic trading.

Keywords: algorithmic trading platform, program code, execution time, repository, instruction and data caches.