Алгоритмы оптимизации машинного кода программ для современных архитектур

Цель: разработка новых подходов к оптимизации исполнения программ без исходных текстов, создание платформы для запуска программ по результатам экспериментов.

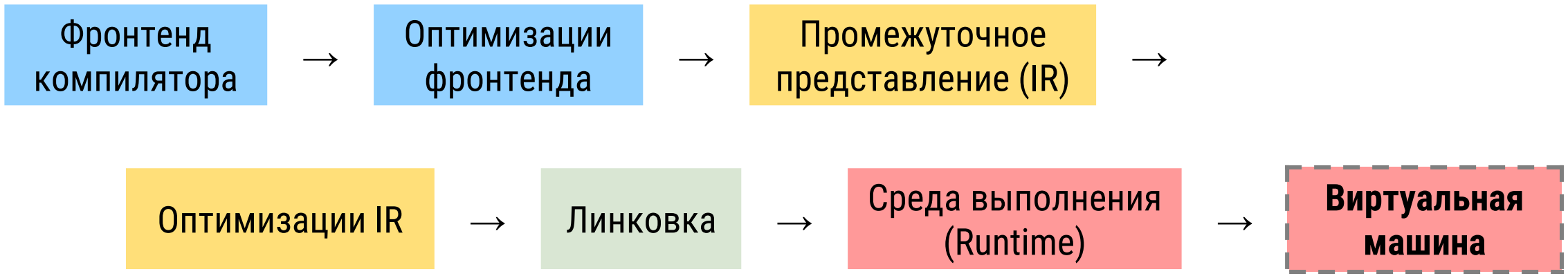
Объект исследования: машинный код программ.

Предмет исследования: временна́я эффективность исполнения машинного кода на различных архитектурах.

Студент гр. 9305: Когутенко Андрей Александрович
Научный руководитель: к. т. н., доцент Пазников Алексей Александрович

Поставленные задачи

1. Рассмотрение работы экспериментальных программ на процессорах с различными микроархитектурами и средствами виртуализации.
2. Разработка новых подходов к оптимизации на основе полученных результатов.
3. Реализация теоретических наработок в платформе, выполняющей трансформации «на лету».



Низкоуровневые оптимизации

1. **Выравнивание кода.**
2. **Разворачивание циклов.**
3. **Избавление от ветвлений, использование CMOV.**
4. Выравнивание данных.
5. SIMD и другие расширения процессора.
6. Избавление от конфликтов данных.
7. На процессорах без внеочередного исполнения – переупорядочивание команд.
8. Уменьшение размера кода для умещения в кэш.
9. . . .

Тестируемые утилиты

Средства трансформации кода в реальном времени:

- QEMU – программа для эмуляции различных платформ. Может как запускать приложения, так и эмулировать машину целиком;
- DynamoRIO – средство динамического анализа программ;
- Valgrind – инструмент для отслеживания утечек памяти и профилирования.

Методика тестирования: многократные прогоны без Turbo Boost, тактовая частота фиксирована.

Исследуемая архитектура: x86-64.

Ассемблер: NASM 2.16.

Компиляторы C/C++: GCC 13.1.1, Clang 15.0.6, ICC 2023.1.0.

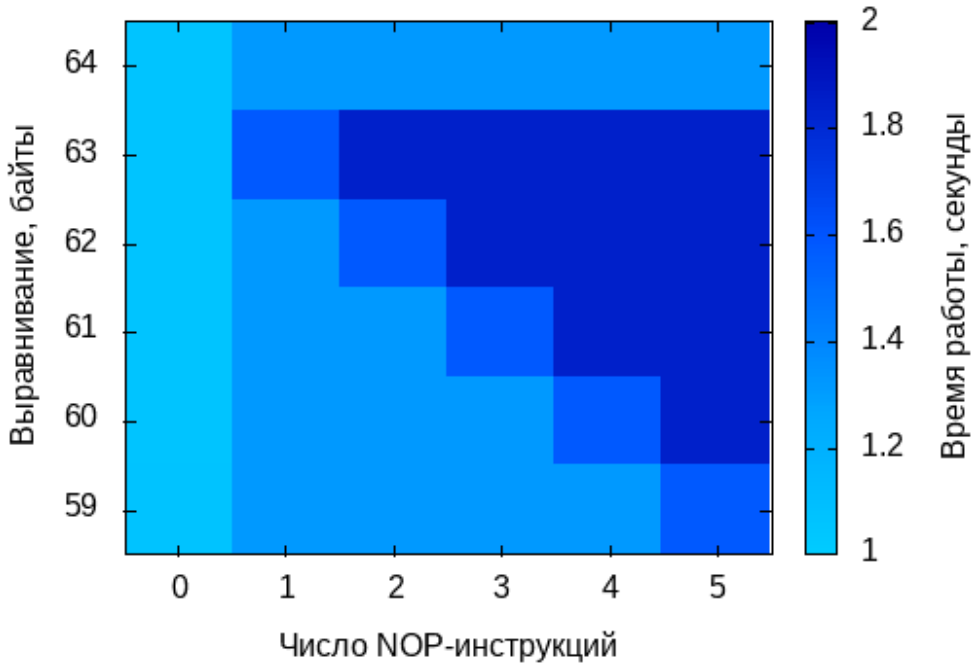
Исследования: выравнивание циклов

```
mov rcx, ITERATIONS
times ALIGNMENT - ($ - _start) nop
loop_begin:
    times INNER_NOPS nop
    loop loop_begin
```

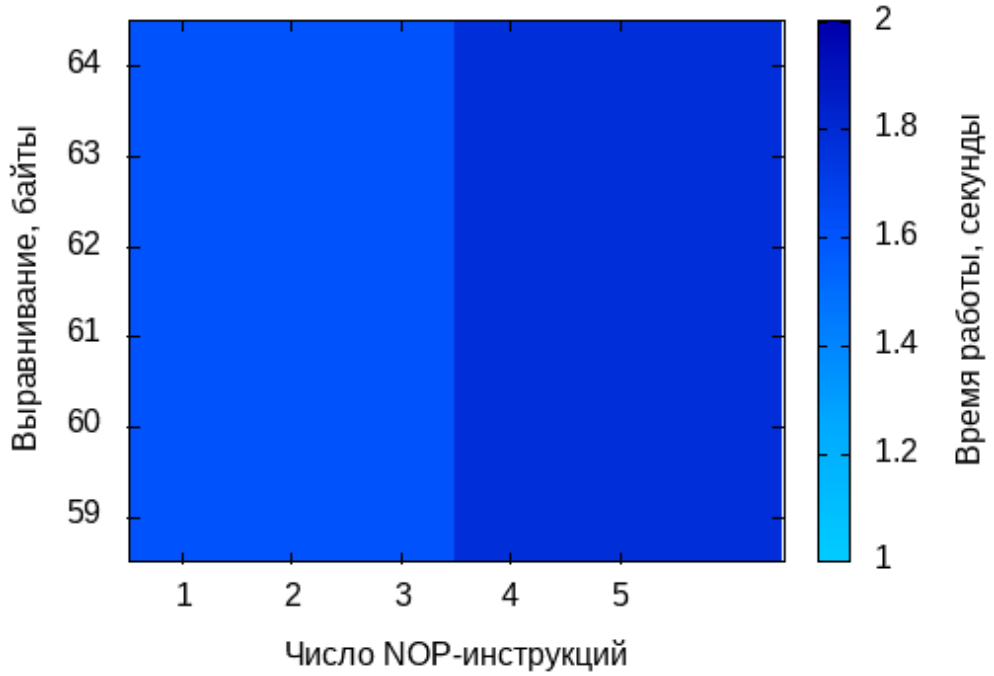
Программа: цикл с NOP-инструкциями.

Идея: цикл, попадающий в разные кеш-линии, может работать медленно.

i7-13700H P-Core 3.8 GHz (Raptor Cove)



i7-13700H E-Core 3.1 GHz (Gracemont)



Исследования: выравнивание циклов

	Невыровненный	QEMU	DynamoRIO	Valgrind	Выровненный
Intel Core i7-13700H, P-Core 3.8 GHz (Raptor Cove)	1,852	0,332	1,854	1,624	1,326
Intel Core i7-13700H, E-Core 3.1 GHz (Gracemont)	1,785	0,982	2,595	2,365	1,784
AMD Ryzen 5 5600H 3.3 GHz (Zen 3)	0,613	0,619	0,624	1,921	0,307
Intel Core i5-10400 2.9 GHz (Comet Lake S)	2,427	20,405	2,400	2,215	1,729
Intel Core i5-9400F 2.9 GHz (Coffee Lake S Refresh)	2,421	20,069	2,445	2,422	1,731
AMD Ryzen 3 3200U 2.7 GHz (Zen+)	0,807	20,473	0,849	3,995	0,389
Intel Celeron J4105 1.5 GHz (Gemini Lake)	7,387	4,083	8,086	10,791	7,393
Intel Pentium 3558U 1.7 GHz (Haswell U)	4,139	38,969	4,170	4,972	2,953

- Практически везде **выравнивание даёт эффект**. Исключения – i7-13700H E-Core, J4105
- QEMU и Valgrind **могут** оптимизировать код

Исследования: разворачивание циклов

Разворачивание циклов – дублирование тела цикла с целью уменьшения числа выполняемых инструкций и ветвлений.

Программа: сложение чисел с развёрнутым циклом.

Идея: 4 вызова инструкции INC могут быть быстрее одного вызова ADD.

```
for (size_t i = 0; i < ARRAY_SIZE; ++i)
    sum += array[i];
```

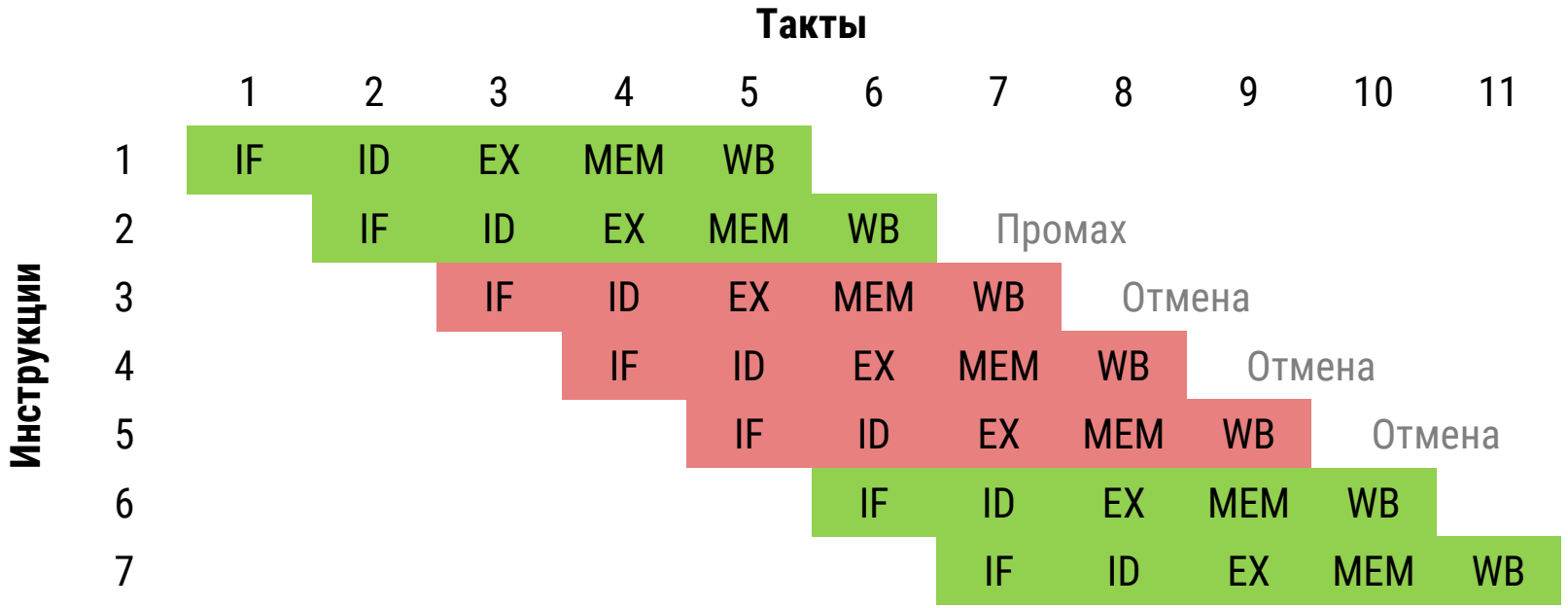
```
loop_begin:
    add rax, [array + rdx * 8 +  0]
    add rax, [array + rdx * 8 +  8]
    add rax, [array + rdx * 8 + 16]
    add rax, [array + rdx * 8 + 24]

%ifdef INC
    inc rdx
    inc rdx
    inc rdx
    inc rdx
%else
    add rdx, 4
%endif

    cmp rdx, ARRAY_SIZE
    jne loop_begin
```

Исследования: разворачивание циклов

	ADD	QEMU	DynamoRIO	Valgrind	4 x INC
Intel Core i7-13700H, P-Core 3.8 GHz (Raptor Cove)	0,653	0,979	0,708	3,077	0,654
Intel Core i7-13700H, E-Core 3.1 GHz (Gracemont)	1,465	1,430	1,598	5,459	1,258
AMD Ryzen 5 5600H 3.3 GHz (Zen 3)	0,596	1,095	0,603	3,894	0,605
Intel Core i5-10400 2.9 GHz (Comet Lake S)	0,902	3,889	1,014	5,486	0,965
Intel Core i5-9400F 2.9 GHz (Coffee Lake S Refresh)	1,217	4,101	1,310	6,413	1,271
AMD Ryzen 3 3200U 2.7 GHz (Zen+)	0,892	3,884	0,956	5,595	0,934
Intel Celeron J4105 1.5 GHz (Gemini Lake)	3,342	5,009	3,450	17,799	3,743
Intel Pentium 3558U 1.7 GHz (Haswell U)	1,513	7,089	1,611	11,398	1,572

- i7-13700H E-Core – 4 инкремента **на 16% быстрее** одного сложения
- QEMU снова **сделал оптимизацию**

Исследования: предсказание ветвлений



При использовании условных переходов процессор делает **предположение**, по какой из ветвей пойдёт исполнение. Если прогноз неудачен, происходит **сброс конвейера**.

Частично эту проблему призвана решить **CMOV** – инструкция условного копирования в регистр. Она **не влияет** на порядок выполнения, тем самым сохраняя эффект применения конвейера.

Исследования: предсказание ветвлений

```
if (value > T)
    ++count;
```

Программа: вычисление количества чисел больше некоторого T.

Идея: реализация с использованием CMOV будет быстрее.

```
%ifdef BRANCHLESS
    ; Вариант без ветвлений
    xor rdx, rdx      ; Обнуляем аккумулятор, который позднее прибавим
    cmp rax, rbx      ; Сравниваем
    cmovg rdx, rdi    ; Устанавливаем аккумулятор в 1, если истинно
    add rsi, rdx      ; Прибавляем к счётчику значение аккумулятора
%else
    ; Стандартная реализация
    cmp rax, rbx      ; Сравниваем
    jle continue      ; Игнорируем инкремент, если число меньше
    inc rsi           ; Инкрементируем счётчик
    continue:
%endif
```

Исследования: предсказание ветвлений

	Оригинал	QEMU	DynamoRIO	Valgrind	CMOV
Intel Core i7-13700H, P-Core 3.8 GHz (Raptor Cove)	0,594	0,625	0,615	0,962	0,212
Intel Core i7-13700H, E-Core 3.1 GHz (Gracemont)	0,609	0,904	0,676	1,476	0,228
AMD Ryzen 5 5600H 3.3 GHz (Zen 3)	0,460	0,708	0,479	1,158	0,207
Intel Core i5-10400 2.9 GHz (Comet Lake S)	0,675	2,533	0,738	1,362	0,284
Intel Core i5-9400F 2.9 GHz (Coffee Lake S Refresh)	0,673	2,738	0,769	1,570	0,277
AMD Ryzen 3 3200U 2.7 GHz (Zen+)	0,633	3,928	0,685	1,773	0,272
Intel Celeron J4105 1.5 GHz (Gemini Lake)	1,438	2,129	1,551	4,222	0,941
Intel Pentium 3558U 1.7 GHz (Haswell U)	1,189	5,291	1,272	2,668	0,591

- На **всех** процессорах реализация с CMOV **быстрее**
- **Никакая** из утилит **не ускорила** оригинал

Исследования: предсказание ветвлений

```
for (uint64_t i = 1; i <= 10000000; ++i)
{
    uint64_t x = i;
    uint64_t iterations = 0;

    while (x != 1 && iterations++ <= 1000)
        if (x % 2 == 0)
            x /= 2;
        else
            x = x * 3 + 1;

    if (iterations == 1000)
        return 1;
}
```

Согласно гипотезе Коллатца, если взять любое натуральное число и выполнить с ним серию операций:

- если чётное – поделить на 2;
- в противном случае умножить на 3 и прибавить 1,

то рано или поздно получим единицу.

Программа: проверка гипотезы Коллатца на числах от 1 до 10^7.

Идея: тестирование компиляторов на применение CMOV.

Исследования: предсказание ветвлений

i7-13700H P-Core 3.8 GHz (Raptor Cove)

	Нативно	QEMU	DynamoRIO	Valgrind
GCC 13.1.1 O2	2,344	4,406	2,860	10,517
Clang 15.0.6 O2	1,589	2,253	1,588	9,070
ICC 2023.1.0 O2	1,568	2,103	1,635	8,794
GCC 13.1.1 Ofast march=native	2,345	4,371	2,861	10,531
Clang 15.0.6 Ofast march=native	1,524	2,931	1,574	9,369
ICC 2023.1.0 Ofast march=native	1,529	2,930	1,574	9,376

- GCC **не применил** CMOV – оказался **хуже** других компиляторов
- **Никакая** из утилит **не ускорила** оригинал

Исследования: зависимость от данных

```
#ifndef DIV
uint32_t div = strtoul(argv[1], NULL, 10);
#endif

for (size_t i = 0; i < ITEMS_COUNT; ++i)
#ifdef DIV
    sum += xorshift32() / DIV;
#else
    sum += xorshift32() / div;
#endif
```

Программа: суммирование псевдослучайных чисел, поделённых на число.

Идея: если число является оптимизируемой константой (например, 16 можно заменить на битовые сдвиги), компилятор применит оптимизацию.

Исследования: зависимость от данных

i7-13700H P-Core 3.8 GHz (Raptor Cove)

	Нативно	**QEMU**	**DynamoRIO**	**Valgrind**
GCC 13.1.1	1,694	2,721	1,706	6,454
Clang 15.0.6	1,691	3,056	1,706	6,333
ICC 2023.1.0	1,653	3,616	1,673	5,480
GCC 13.1.1 const	1,745	2,657	1,757	4,563
Clang 15.0.6 const	1,643	2,315	1,644	4,017
ICC 2023.1.0 const	1,642	2,313	1,648	3,713

- GCC работает **хуже**… при **константе**!
- При замене «/» на «%» GCC работает как надо
- Компилятор от Intel выдал **более быстрый** код
- **Никакая** из утилит **не ускорила** оригинал

Исследования: сортировка методом пузырька

```
for (size_t i = 1; i < ITEMS_COUNT; ++i)
    for (size_t j = 1; j <= ITEMS_COUNT - i; ++j)
        if (array[j - 1] > array[j])
        {
            int temp = array[j - 1];
            array[j - 1] = array[j];
            array[j] = temp;
        }
```

Программа: сортировка пузырьком 50 тысяч целых чисел.

Идея: тестирование компиляторов.

Исследования: сортировка методом пузырька

i7-13700H P-Core 3.8 GHz (Raptor Cove)

	Нативно	QEMU	DynamoRIO	Valgrind
GCC 13.1.1	3,997	9,734	4,382	27,209
Clang 15.0.6	4,066	8,161	4,381	28,448
ICC 2023.1.0	3,943	7,723	4,280	28,551
GCC 13.1.1 O2	5,790	10,828	5,694	35,434
Clang 15.0.6 O2	2,217	4,688	2,414	10,330
ICC 2023.1.0 O2	2,610	4,066	2,613	10,682
GCC 13.1.1 O2 fno-tree-vectorize	3,019	4,443	3,295	11,927
Clang 15.0.6 O2 fno-tree-vectorize	2,221	4,686	2,418	10,290
ICC 2023.1.0 O2 fno-tree-vectorize	2,613	4,065	2,613	10,696

- При O2 код GCC работает **хуже**, чем при O0. Отключение векторизации не сильно спасает
- Код ICC оказался **медленнее** Clang
- **Никакая** из утилит **не ускорила** оригинал

Исследования: сортировка методом пузырька

```
outer_loop:
    mov r9, 1          ; Храним значение переменной j
    inner_loop:
        mov edx, [rdi + r9 * 4]          ; arr[j]
        mov eax, [rdi + r9 * 4 - 4]      ; arr[j - 1]
        mov r10d, edx                    ; Храним значение одной из переменных
                                         ; на случай, если придётся делать обмен
        cmp eax, edx                     ; Сравниваем
        cmovg edx, eax                   ; Меняем местами, если больше
        cmovg eax, r10d
        mov [rdi + r9 * 4], edx          ; Пишем arr[j - 1] в arr[j]
        mov [rdi + r9 * 4 - 4], eax      ; Пишем arr[j] в arr[j - 1]
        inc r9                           ; Инкрементируем j
        cmp r9, r8                       ; Сравниваем с предельным значением
        jl inner_loop                    ; Если не достигли предела, идём в начало
    dec r8              ; Уменьшаем внешний счётчик
    test r8, r8         ; Проверяем на ноль
    jnz outer_loop      ; Если не ноль, переходим в начало внешнего цикла
```

Исследования: сортировка методом пузырька

	clang Ofast	nasm
Intel Core i7-13700H, P-Core 3.8 GHz (Raptor Cove)	2,097	2,757
Intel Core i7-13700H, E-Core 3.1 GHz (Gracemont)	2,873	1,223
AMD Ryzen 5 5600H 3.3 GHz (Zen 3)	2,193	0,876
Intel Core i5-10400 2.9 GHz (Comet Lake S)	3,390	2,587
Intel Core i5-9400F 2.9 GHz (Coffee Lake S Refresh)	3,800	2,607
AMD Ryzen 3 3200U 2.7 GHz (Zen+)	3,751	4,985
Intel Celeron J4105 1.5 GHz (Gemini Lake)	6,558	7,559
Intel Pentium 3558U 1.7 GHz (Haswell U)	5,485	6,754

- E-ядро… **быстрее на 70%**?..
- Практически на всех **современных** процессорах ассемблерная версия **быстрее**
- PGO оказался **бесполезен**

Исследования: заключение

1. Иногда компиляторы не применяют очевидные оптимизации – например, использование CMOV.
2. Микроархитектура может оказывать значительное влияние на эффективность кода.
3. Ни одна из инструментирующих утилит не даёт гарантии минимальных накладных расходов.
4. Программа может быть адаптирована в соответствии с входными данными и работать быстрее.
5. PGO не всегда даёт хорошие результаты.

- Можно создать **динамический оптимизатор**, который будет адаптировать «на лету»
- **Проблема** – как избавиться от накладных расходов?

Алгоритмы оптимизации: новые подходы

Некоторые соображения:

- основное процессорное время приходится на циклы;
- в программе есть лишь несколько высоконагруженных мест – важно, чтобы именно они работали быстро;
- необходимо избегать работы инструментирующей программы во время работы циклов;
- работа инструментирования должна быть максимально быстрой;
- при генерации машинного кода нужно учитывать особенности микроархитектуры;
- машинный код можно адаптировать к входным данным.

Алгоритмы оптимизации: идея 1

```
...
xor rax, rax
xor rbx, rbx
xor rcx, rcx

iterate:
    cmp [array + rcx * 8], rbx
    jle continue
    inc rax
    continue:
    inc rcx
    cmp rcx, ARRAY_SIZE
    jne iterate

bswap rax
mov [count], rax
...
```

Точка возврата управления – вставка трансформирующей среды для возврата контроля.

Точки возврата управления необходимо ставить **вне нагруженных мест**, которыми являются циклы.

Алгоритмы оптимизации: идея 2

```
measure_start();
for (0..100) <transform1>
measure_reset();
for (100..200) <transform2>
measure_reset();
 ...
measure_reset();
for ((N-1)*100..N*100) <transformN>
measure_end();

for (N*100..10000) <transformBest>
```

Апробация трансформаций – генерация нескольких эквивалентных вариантов машинного кода и испытание каждого на малом числе итераций для выбора лучшего.

Лучшие трансформации можно **кешировать**.

Алгоритмы оптимизации: идея 3

Предварительный просчёт – запуск оптимизатора отдельно от программы для исследования характеристик системы.

Позволяет **сократить** время на трансформацию при работе программы.

Заранее можно:

- вывести всевозможные арифметические выражения, сгенерировать и опробовать их эквиваленты в машинном коде;
- узнать время работы отдельных инструкций и их комбинаций, оценить их работу в различных контекстах;
- протестировать различные паттерны работы с памятью.

Практика: тестер микроархитектуры

Разработано приложение для **тестирования микроархитектуры**.

Что делает: проверяет различные паттерны машинного кода.

Для чего: помощь программисту в исследовании возможностей целевой платформы.

Loop alignment (LOOP)

	0	1	2	3	4	5
59:	1.06	1.32	1.32	1.32	1.32	1.58
60:	1.06	1.32	1.32	1.32	1.58	1.85
61:	1.06	1.32	1.32	1.58	1.85	1.85
62:	1.06	1.32	1.58	1.85	1.85	1.85
63:	1.06	1.58	1.85	1.85	1.85	1.85
64:	1.06	1.32	1.32	1.32	1.32	1.32

Unaligned sequential read

	0	1	2	3	4	5	6	7	8
L1:	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
L2:	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
L3:	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29
Mem:	0.46	0.46	0.46	0.46	0.46	0.46	0.46	0.46	0.46

Практика: тестер микроархитектуры

```
bin = push(bin, "\x48\x89\xF9");          // mov rcx, rdi

// Alignment
for (size_t i = 3; i < alignment; ++i)
    bin = push(bin, "\x90");              // nop

// NOPs in loop
for (size_t i = 0; i < nop_count; ++i)
    bin = push(bin, "\x90");              // nop

if (loop) {
    bin = push(bin, "\xE2");              // loop loop_begin
    *bin++ = 0 - nop_count - 2;
} else {
    bin = push(bin, "\x48\xFF\xC9");      // dec rcx
    bin = push(bin, "\x75");              // jnz loop_begin
    *bin++ = 0 - nop_count - 3 - 2;
}
```

Практика: платформа для трансформаций

Общий алгоритм работы:

1. Чтение начала исполняемого файла и разбор ELF-заголовка.
2. Загрузка сегментов в виртуальную память по указанным в ELF адресам.
3. Анализ и трансформация машинного кода алгоритмами, не требующими апробации.
4. Генерация эквивалентных вариантов кода и модификация следующего цикла.
5. Вставка точки возврата управления перед циклом.
6. Загрузка модифицированной программы в память.
7. Вызов программы.
8. Возврат управления программой.
9. Если программа не завершила работу, переход к п. 3.

Практика: платформа для трансформаций

Уже реализованные оптимизации:

- удаление бессмысленного кода – без апробации;
- выравнивание циклов – без апробации;
- использование обычного сложения (ADD) или инкремента (INC) – с апробацией.

Реализующиеся в настоящее время:

- трансформация ветвлений с использованием CMOV;
- учёт значений регистров при входе в цикл;
- предварительный просчёт – арифметические выражения и эквиваленты.

Практика: платформа для трансформаций

Технические сложности:

1. Релокация в случае, когда адреса сегментов вызываемой программы пересекаются с адресами оптимизатора.
2. Определение точек возврата управления внутри циклов, когда это допустимо.
3. Грамотная постановка предпросчёта.
4. Учёт аргументов системных вызовов в контексте меняющегося ABI.

Практика: платформа для трансформаций

```
somebody@Somebody:~/binboost$ cat nop-loop.asm
global _start

_start:

    mov rcx, 1000000000     ; Iterations count

    loop_begin:             ; Useless loop
        nop
        nop
        nop
        nop
        loop loop_begin

    mov rax, 60             ; Terminate
    xor rdi, rdi
    syscall
somebody@Somebody:~/binboost$ nasm -f elf64 nop-loop.asm
somebody@Somebody:~/binboost$ ld -o nop-loop nop-loop.o -Ttext 0x1000000
somebody@Somebody:~/binboost$ time ./nop-loop
0.434
somebody@Somebody:~/binboost$ time qemu-x86_64 ./nop-loop
0.859
somebody@Somebody:~/binboost$ time ./binboost ./nop-loop
0.014
```

Преимущества подхода:

1. Можно запускать программы без исходного кода – особенно актуально для старых приложений.
2. Возможность адаптировать программу к входным данным.
3. Учёт микроархитектурных особенностей.
4. Генерация трансформаций может опираться на результаты предыдущих проб.

Проект НИР с Huawei

Тема: «Средства оптимизации распределения данных»

Личный вклад:

- анализ производительности трансформаций рекурсивных и нерекурсивных структур;
- новый аллокатор для более эффективного использования кэш-памяти;
- две идеи оптимизации трансформаций, демонстрация их эффективности;
- переборные алгоритмы для поиска лучших разбиений структур, их реализация;
- разработка алгоритмов разбиения структур на основе профиля;
- скрипт для применения трансформаций;
- эксперименты на собственных тестах и бенчмарках SPEC.

Заключение

1. Проведены исследования различных оптимизаций на различных процессорах.
2. Разработаны новые подходы к оптимизации.
3. Создано приложение для тестирования микроархитектур.
4. Реализована простейшая платформа для эффективных оптимизаций «на лету».
5. Для созданных программ планируется дальнейшее развитие и поддержка.
6. Полученные результаты дорабатываются для публикации в журнале и регистрации ПО.