СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В 2019 году процессоры архитектуры ARM составляли 34% от рынка процессоров, при этом занимая 90% рынка мобильных процессоров [1]. С появлением процессоров M1 компании Apple большое число людей стали пользоваться компьютерами на основе архитектуры ARM в качестве персональных компьютеров, часто у пользователей появляется необходимость использовать программное обеспечение с закрытым исходным кодом, что не позволяет их перекомпилировать под необходимую архитектуру самостоятельно.

Для запуска программ собранных под архитектуру x86 необходим или статический транслятор, такой как Rosetta 2, или виртуальная машина поддерживающая необходимую архитектуру, или динамический транслятор для пользовательского режима. Rosetta 2 не транслирует программы не предназначенные для macOS, для запуска программ созданных для Windows или Linux нужно использовать виртуальную машину. Еще одно ограничение статической трансляции — наличие самомодифицирующегося кода и динамических библиотек, таким образом использование только статической трансляции не запустит любую программу. [2]

Использование трансляции по своей природе несет дополнительные издержки при выполнении программы, как в памяти, так и в скорости выполнения. Цель этой работы — исследование причин дополнительных издержек и попытка нахождения пути устранения этих издержек.

Для достижения поставленной цели необходимо решить следующие задачи:

— реализовать оптимизирующий проход над промежуточным представлением;
— встроить полученную реализацию в динамический транслятор FEX;
— исследовать результаты работы оптимизирующего прохода;
— исследовать корректность работы оптимизирующего прохода.

1 Технологическая часть

В данном разделе описываются средства разработки программного обеспечения, требования к вычислительной системе. Приводится структура разработанного приложения.

1.1 Выбор средств разработки

Из рассмотренных в аналитическом разделе трансляторов FEX лучше прочих подходит для реализации алгоритма, так как в нем присутствует СЕП.

1.1.1 Выбор языка программирования

Динамический транслятор FEX написан на языке C++, в нем используются разные библиотеки, например ядро транслятора FEXCore это библиотека реализующая непосредственно динамическую трансляцию. FEXCore также написан на C++ и не предусматривает расширений с помощью внешних модулей, поэтому для того чтобы ее модифицировать надо писать код на C++.

1.1.2 Сборка программного обеспечения

Для сборки проекта используется система сборки CMake. Для компиляции реализации алгоритма необходимо добавить имя файла в соответствующий CMakeLists.txt.

Для сборки всего проекта используется ninja. На листинге 3 указана конфигурация сборки проекта.

Листинг 1: Конфигурация сборки проекта

```
1  CC=clang CXX=clang++ cmake -DCMAKE_INSTALL_PREFIX=/usr \
2  -DCMAKE_BUILD_TYPE=Release -DENABLE_LTO=True \
3  -DBUILD_TESTS=False -DCMAKE_CXX_COMPILER_LAUNCHER=ccache \
4  -DENABLE_CLANG_FORMAT=False -G Ninja ..
```

Оптимизирующий проход создан на основе версии 2204, коммит 37f1e55ed5dc7a35ba9bf875e250de0b75581a22.

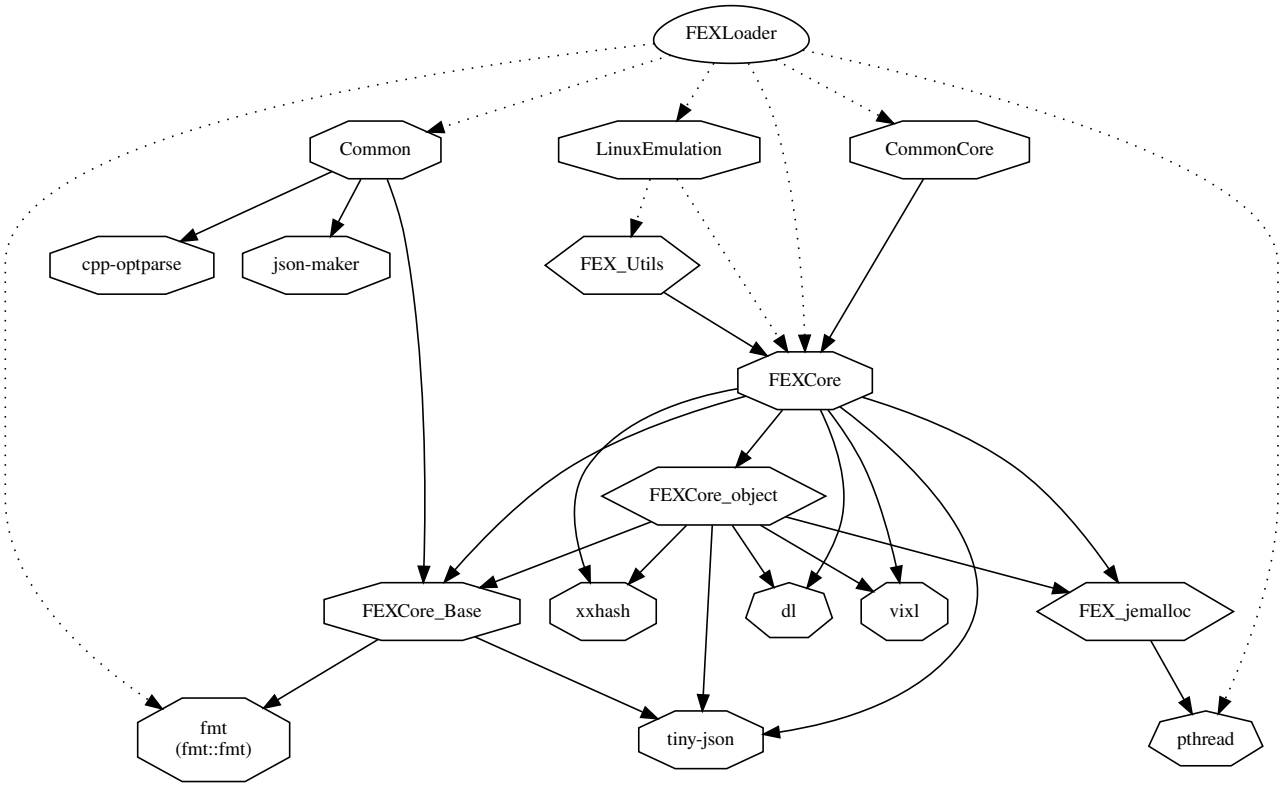

Рисунок 1 – Диаграмма зависимостей userspace эмулятора FEXLoader

Вносимые в проект изменения относятся к FEXCore_Base.

1.2 Требования к вычислительной системе

Для запуска программного обеспечения требуется исходный код динамического транслятора FEX. Так как алгоритм реализует оптимизацию доступа к памяти с сохранением корректного выполнения многопоточных приложений для прироста в скорости требуется процессор с поддерживаемой архитектурой со слабым доступом к памяти и несколькими ядрами, то есть оптимизирующий проход будет приводить к росту производительности не только на ARM, но и, например, RISC-V.

Для разработки и сборки проекта использовался ноутбук на основе процессора RK3399, архитектуры ARM. Несмотря на это разработка не привязана к конкретной архитектуре, теоретически разработку можно вести на любой архитектуре имеющий набор инструментов для кросс-компиляции под архитектуру ARM.

1.3 Структура программного обеспечения

Разработанное ПО реализовано как оптимизирующий проход над промежуточным представлением.

При инициализации транслятора вызывается функция AddDefaultPasses, внутри которой вызываются функции InsertPass регистрирующие оптимизирующие проходы в определенном порядке. Функция InsertPass принимает на вход std::unique_ptr<Pass> — указатель на экземпляр класса прохода. У класса обязательно должен быть метод Run принимающий на вход IREmitter и возвращающий bool — был ли изменен код во время прохода.

Разработанный проход зависит от StaticRegisterAllocationPass, во время этого прохода статически выделяются регистры, на основе этих регистров и идет оптимизация доступа памяти.

Отключить все оптимизирующие проходы можно с помощью аргумента -O или –o0.

1.4 Распространение программного обеспечения

Программное обеспечение распространяется в виде патча сформированного командой git format-patch. Для применения патча к проекту можно, например, использовать git am.

1.5 Дополнительные утилиты

Для анализа генерируемого кода нужно анализировать скомпилированный код, такой возможности у FEX нет, поэтому была проведена модификация функции Context::CompileBlock которая компилирует и запускает блок. Перед запуском вся область запускаемой памяти сохраняется в бинарный файл.

Листинг 2: Сохранение скомпилированного блока

```
--- a/External/FEXCore/Source/Interface/Core/Core.cpp
+++ b/External/FEXCore/Source/Interface/Core/Core.cpp
@@ -1001,6 +1001,15 @@ namespace FEXCore::Context {
}
}

+
+    FILE *fp;
+
+    std::string filename = "Core_Block_other_" + \
+      std::to_string((uint64_t)CodePtr) + ".cdmp";
+    fp = fopen(filename.c_str(), "wb");
+    fwrite((void*)CodePtr, DebugData->HostCodeSize, 1, fp);
+    fclose(fp);
+    printf("block dumped %s\n", filename.c_str());
+
if (IRCaptureCache.PostCompileCode(
Thread,
CodePtr,
--
2.36.0

```

Обработка сохраненной памяти затем выполнялась при помощи objdump — в итоге получался код на ассемблере.

После анализа скомпилированного кода были сделаны выводы по основным факторам замедляющим выполнение скомпилированного кода, таких как барьерный доступ к памяти.

1.6 Вывод

В данном разделе были описаны средства разработки программного обеспечения, требования к вычислительной системе. Была дана структура разработанного приложения.

2 Исследовательская часть

В рамках дипломной работы было проведено исследование изменения результатов бенчмарка nbench с разработанным проходом. Результаты исследования представлены в этом разделе.

2.1 Описание используемых данных

Для проведения исследования используется бенчмарк nbench скомпилированный при помощи gcc в два разных бинарных файла: с флагом -O0 и с флагом -O3.

В бенчмарке nbench реализовано 9 тестов:

— Numeric sort — сортировка массива 32-х битных чисел. Оценивается производительность работы с целыми числами.

— String sort — сортировка массива символов. Проверяется скорость работы с памятью.

— Bitfield — выполняет различные битовые операции. Например: очистить или выставить n бит, инвертировать число. Оценивается скорость выполнения таких операций.

— Emulated floating-point — небольшой эмулятор для работы с числами с плавающей запятой. Хорошо оценивает общую производительность.

— Fourier coefficients — рассчитывает коэффициенты Фурье. Оценивает производительность FPU, при этом память использует не активно.

— Assignment algorithm — решение задачи о назначениях. Работает с массивом, на результат влияет скорость последовательного доступа к памяти.

— Huffman compression — алгоритм Хаффмана. Операции с байтами, битами и с целыми числами. Считается хорошей оценкой общей производительности.

— IDEA encryption — алгоритм шифрования. Оценивает скорость исполнения кода.

— LU Decomposition — алгоритм решения линейных уравнений. Работает с числами с плавающей запятой, использует только фундаментальные операции (+, -, *, /).

В итоге получаются INTEGER INDEX и FLOATING-POINT INDEX — комбинированный результат бенчмарков.

Каждый бенчмарк запускается 5 раз, затем рассчитывается среднеквадратическое отклонение и доверительный интервал с уровнем доверия 0.95. Если длина полуинтервала меньше 1% от абсолютного значения среднего, то замер времени считается успешным, иначе проводятся дополнительные замеры.

Динамический транслятор FEX запускается на SOC Rockchip RK3399, Exynos 8895 и Apple M1. Система работающая на RK3399 работает под управлением Linux; Exynos — Android, сам запуск производится через chroot в окружение Debian; Apple M1 — виртуальная машина с Ubuntu.

Из-за того что SOC RK3399 и Exynos 8895 используют архитектуру big.LITTLE у них одинаковые «малые» ядра — Cortex A53. Больших различий между этими ядрами нет, поэтому в тестах использовались результаты работы этого ядра на RK3399. Раздельно представлены результаты бенчмарков для их «больших» ядер — Cortex A72 для RK3399 и Mongoose M2 для Exynos 8895.

Замеры времени выполнения проводились на версии FEX-2204.

2.2 Результаты исследования

На рисунках 2-6 представлены результаты замеров скорости выполнения nbench с использованием алгоритма и без.

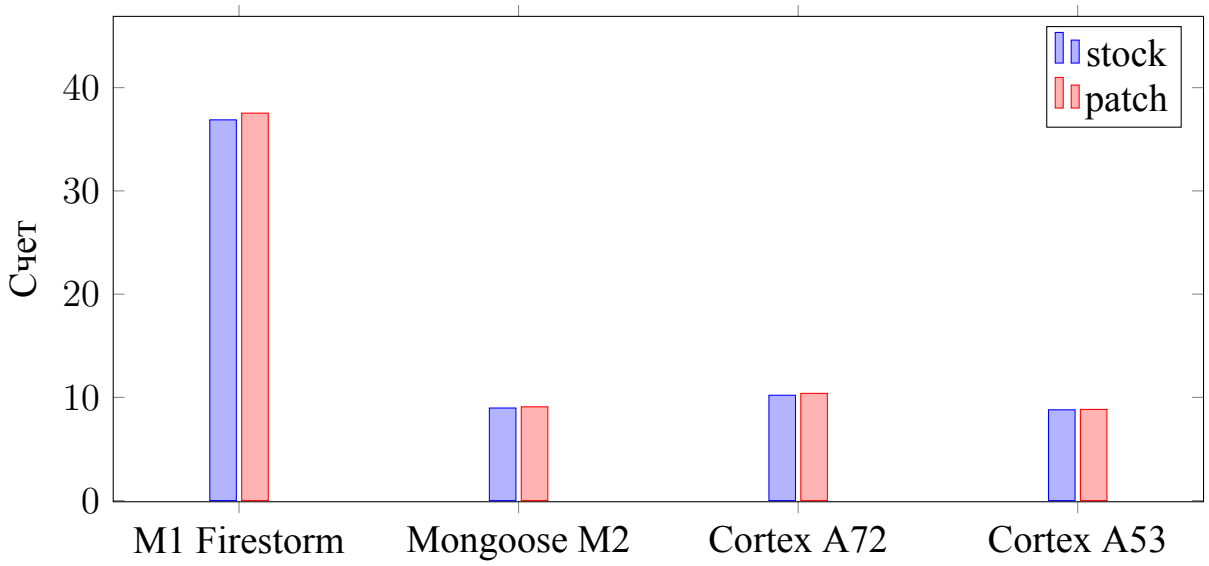

Рисунок 2 – nbench O0

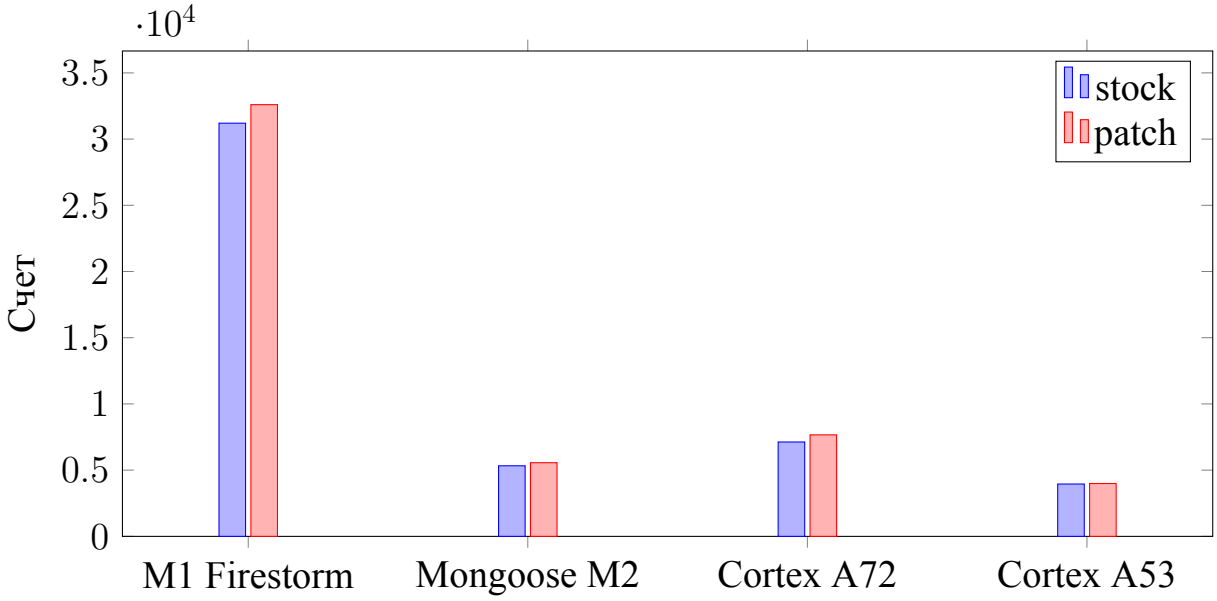

Рисунок 3 – nbench FOURIER O0

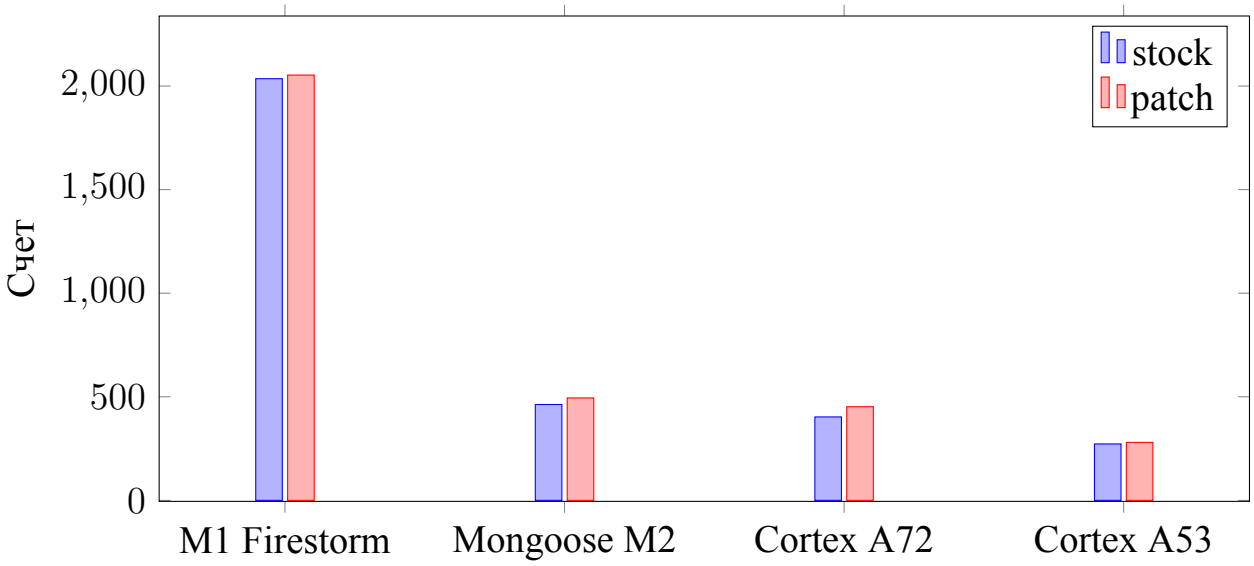

Рисунок 4 – nbench IDEA O0

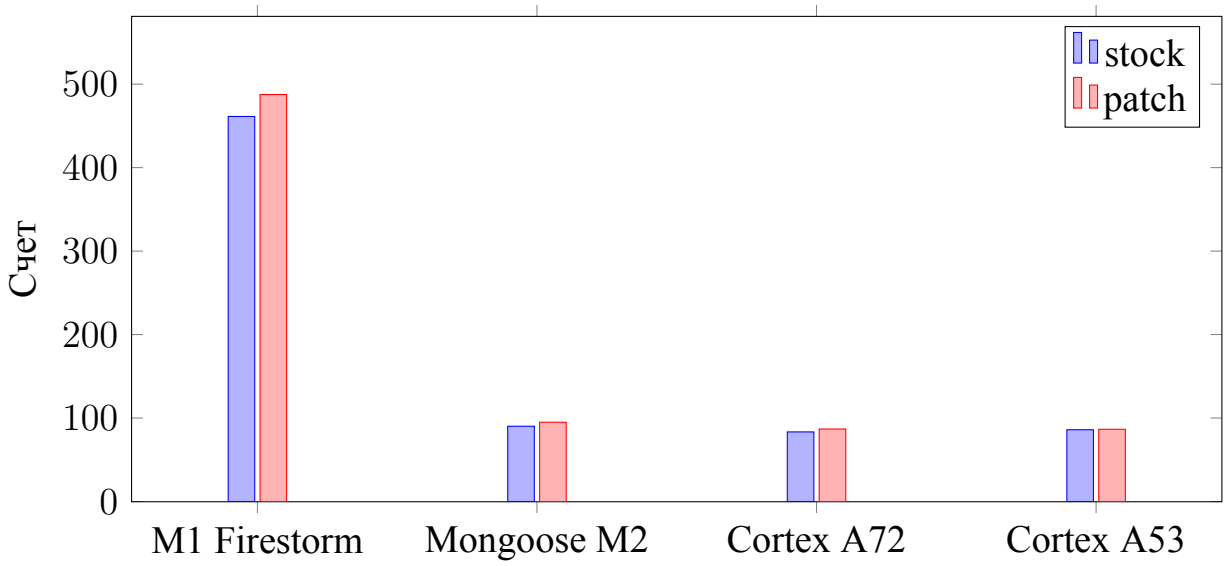

Рисунок 5 – nbench HUFFMAN O0

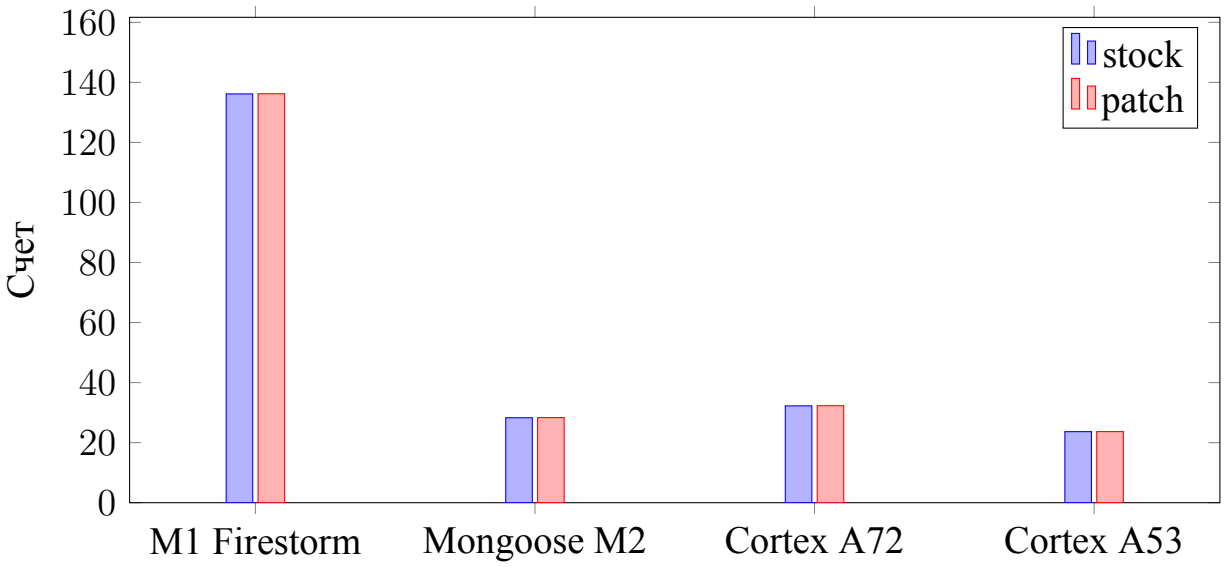

Рисунок 6 – nbench O3

Таким образом по результатам замеров видно что проход работает только на не оптимизированном бенчмарке (собранном с флагом -O0), также стоит заметить что для Apple M1 рост в производительности на выделенных бенчмарках меньше.

На O3 рост производительности незначителен, это связано с тем что регистр RBP, при оптимизации, используется в качестве регистра общего назначения.

Рост в производительности составил:

— Cortex A53 — производительность в среднем выше в 0.42 процента, для FOURIER — прирост 1 процент, IDEA — 2.7 процента, HUFFMAN — меньше процента;

— Cortex A72 — производительность в среднем выше в 1.77 процента, для FOURIER — прирост 7.5 процента, IDEA — 12.2 процента, HUFFMAN — 4.2 процента;

— Mongoose M2 — производительность в среднем выше в 1.29 процента, для FOURIER — прирост 4.4 процента, IDEA — 6.8 процента, HUFFMAN — 5.3 процента;

— M1 Firestorm — производительность в среднем выше в 1.76 процента,

для FOURIER — прирост 4.5 процента, IDEA — меньше процента, HUFFMAN — 5.7 процента;

Более высокий рост производительности для старых ядер связан с затратностью барьерных операций с памятью, новые поколения ядер уменьшают затратность таких операций. У Cortex A53, как было выяснено в аналитическом разделе, разница в скорости выполнения программ с барьерным доступом к памяти и без него мала, поэтому для этого ядра результаты меньше.

2.3 Тестирование на надежность

Главной причиной разработки алгоритма по оптимизации доступа к памяти являлась слабая модель архитектуры ARM, что при трансляции многопоточных приложений приводило к ошибкам.

Для проверки корректной работы прохода использовался tst-cond16.c — тест библиотеки libc. Этот тест создает 8 потоков которые проводят операции с мьютексами. Стандартная версия транслятора и версия с оптимизирующим проходом этот тест проходит, без барьерных инструкций и с безусловной заменой барьерных операций связанных с регистром RBP тест не проходит.

2.4 Вывод

В результате исследования было установлено, что проход увеличивает производительность отдельных бенчмарков и в целом повышает скорость выполнения транслированного кода, но только если исходный код был собран без оптимизаций.

Также было частично доказано что барьерные инструкции доступа к памяти сильнее влияют на ранее выпущенные ядра ARM.

ЗАКЛЮЧЕНИЕ

В рамках этой работы:

— был реализован оптимизирующий проход над промежуточным представлением;

— проход был встроен в динамический транслятор FEX;

— проведено исследование результатов работы оптимизирующего прохода;

— проведено исследование корректность работы оптимизирующего прохода.

Таким образом цель работы — нахождения и устранение издержек трансляции была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Arm Limited Roadshow Slides Q1 2020 [Электронный ресурс]. – Режим доступа: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q1fy2020_01_en.pdf, свободный – (24.11.2021)

2. Probst, Mark. Fast Machine-Adaptable Dynamic Binary Translation. 2001.

ПРИЛОЖЕНИЕ А

Листинг 3: Проход оптимизации доступа к памяти

```
#include <FEXCore/Core/X86Enums.h>
#include <FEXCore/IR/IR.h>
#include <FEXCore/IR/IREmitter.h>
#include <FEXCore/IR/IntrusiveIRList.h>

#include <array>
#include <iostream>

#include "Interface/Core/OpcodeDispatcher.h"
#include "Interface/IR/PassManager.h"

namespace FEXCore::IR {

    enum RBP_STATE {
        NOT_CHANGED = (0b000 << 0),
        STACK = (0b001 << 0),
        NOT_STACK = (0b010 << 0),
    };

    struct BlockInfo {
        int State = NOT_CHANGED;
        std::set<OrderedNode*> StackNodes;
        std::set<OrderedNode*> UnStackNodes;
        std::vector<OrderedNode*> Predecessors;
        bool Visited = false;
    };

    class StackAccessTSORemovalPass final : public Pass {
        public:
        bool Run(IREmitter* IREmit) override;

        private:
        void CalculateControlFlowInfo(IREmitter* IREmit);
        int CalculateComplexState(const OrderedNode* TargetNode,
        const IRListView& CurrentIR);
        std::unordered_map<NodeID, BlockInfo> OffsetToBlockMap;
    };

    static bool HitsRegister(const int Register, IREmitter* IREmit,
    IROp_Header* Op) {
        std::queue<IROp_Header*> values{};
        values.push(Op);

        while (!values.empty()) {
```

```
45                IROp_Header* ValOp = values.front();
46                values.pop();
47
48                switch (ValOp->Op) {
49                    case IR::OP_ADD:
50                    case IR::OP_SUB:
51                    case IR::OP_OR:
52                    case IR::OP_AND: {
53                        values.push(IREmit->GetOpHeader(ValOp->Args[0]));
54                        values.push(IREmit->GetOpHeader(ValOp->Args[1]));
55
56                        break;
57                    }
58                    case IR::OP_LOADREGISTER: {
59                        auto LocalOp = ValOp->CW<IR::IROp_LoadRegister>();
60
61                        if (LocalOp->Offset ==
62                        offsetof(FEXCore::Core::CPUState, gregs[Register]) &&
63                        LocalOp->Class == GPRClass) {
64                            return true;
65                        }
66
67                        break;
68                    }
69                    default: {
70                        break;
71                    }
72                }
73            }
74
75            /* never hit the RSP register, so it's not a stack value being
76            * loaded. */
77            return false;
78        }
79
80        void StackAccessTSORemovalPass::CalculateControlFlowInfo(IREmitter* IREmit) {
81            IRListView CurrentIR = IREmit->ViewIR();
82
83            for (auto [BlockNode, BlockHeader] : CurrentIR.GetBlocks()) {
84                BlockInfo* CurrentBlock =
85                &OffsetToBlockMap.try_emplace(CurrentIR.GetID(BlockNode)).first->second;
86                for (auto [CodeNode, IROp] : CurrentIR.GetCode(BlockNode)) {
87                    switch (IROp->Op) {
88                        case IR::OP_CONDJUMP: {
89                            auto Op = IROp->CW<IR::IROp_CondJump>();
90
91                            {
92                                auto Block =
93                                &OffsetToBlockMap.try_emplace(Op->TrueBlock.ID()).first->second;
94                                Block->Predecessors.emplace_back(BlockNode);
95                            }
```

```
96
97                              {
98                                  auto Block = &OffsetToBlockMap.try_emplace(Op->FalseBlock.ID())
99                                  .first->second;
100                                 Block->Predecessors.emplace_back(BlockNode);
101                             }
102
103                             break;
104                         }
105                         case IR::OP_JUMP: {
106                             auto Op = IROp->CW<IR::IROp_Jump>();
107
108                             {
109                                 auto Block =
110                                 OffsetToBlockMap.try_emplace(Op->Header.Args[0].ID()).first;
111                                 Block->second.Predecessors.emplace_back(BlockNode);
112                             }
113
114                             break;
115                         }
116                         case IR::OP_STOREREGISTER: {
117                             IROp_StoreRegister* Op = IROp->CW<IR::IROp_StoreRegister>();
118                             if (Op->Offset == offsetof(FEXCore::Core::CPUState,
119                             gregs[FEXCore::X86State::REG_RBP]) &&
120                             Op->Class == GPRClass) {
121                                 if (HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
122                                 IREmit->GetOpHeader(Op->Value))) {
123                                     CurrentBlock->State = STACK;
124                                     CurrentBlock->StackNodes.emplace(CodeNode);
125                                 } else {
126                                     CurrentBlock->State = NOT_STACK;
127                                     CurrentBlock->UnStackNodes.emplace(CodeNode);
128                                 }
129                             }
130                             break;
131                         }
132                         default:
133                         break;
134                     }
135                 }
136             }
137         }
138
139         // returns the combined state of all the predecessors and the block itself
140         int StackAccessTSORemovalPass::CalculateComplexState(
141         const OrderedNode* TargetNode, const IRListView& CurrentIR) {
142             auto TargetPair = OffsetToBlockMap.find(CurrentIR.GetID(TargetNode));
143             if (TargetPair == OffsetToBlockMap.end()) {
144                 std::cout << "TargetPair not found in offsets!" << std::endl;
145                 return NOT_STACK;
146             }
```

```
147
148             BlockInfo* TargetBlock = &TargetPair->second;
149
150             if (TargetBlock->Visited || TargetBlock->Predecessors.size() == 0 ||
151             TargetBlock->State != NOT_CHANGED) {
152                 return TargetBlock->State;
153             }
154
155             TargetBlock->Visited = true;
156             int ResultingState = STACK;
157             for (auto i : TargetBlock->Predecessors) {
158                 // only call if predecessor is not visited?
159                 if (i == TargetNode) {
160                     continue;
161                 }
162                 int PredecessorState = CalculateComplexState(i, CurrentIR);
163
164                 if (PredecessorState == NOT_CHANGED) {
165                     ResultingState = NOT_CHANGED;
166                 } else if (PredecessorState == NOT_STACK) {
167                     ResultingState = NOT_STACK;
168                     break;
169                 }
170             }
171
172             TargetBlock->State = ResultingState;
173
174             return TargetBlock->State;
175         }
176
177         bool StackAccessTSORemovalPass::Run(IREmitter* IREmit) {
178             CalculateControlFlowInfo(IREmit);
179
180             IRListView CurrentIR = IREmit->ViewIR();
181             bool Changed = false;
182
183             for (auto [BlockNode, BlockHeader] : CurrentIR.GetBlocks()) {
184                 auto CurrentBlockPair = OffsetToBlockMap.find(CurrentIR.GetID(BlockNode));
185                 if (CurrentBlockPair == OffsetToBlockMap.end()) {
186                     std::cout << "Block not found in offsets" << std::endl;
187                     continue;
188                 }
189
190                 int StackStatus = STACK;
191                 BlockInfo* CurrentBlock = &CurrentBlockPair->second;
192                 for (auto CodeNode : CurrentBlock->Predecessors) {
193                     int PredecessorState = CalculateComplexState(CodeNode, CurrentIR);
194
195                     if (PredecessorState != STACK) {
196                         StackStatus = PredecessorState;
197                         break;
```

```
198                     }
199                 }
200
201                 for (auto [CodeNode, IROp] : CurrentIR.GetCode(BlockNode)) {
202                     if (CurrentBlock->StackNodes.contains(CodeNode)) {
203                         StackStatus = STACK;
204                     }
205                     if (CurrentBlock->UnStackNodes.contains(CodeNode)) {
206                         StackStatus = NOT_STACK;
207                     }
208                     if (IROp->Op == IR::OP_LOADMEMTSO && StackStatus == STACK) {
209                         if (HitsRegister(FEXCore::X86State::REG_RBP, IREmit,
210                         IREmit->GetOpHeader(IROp->Args[0])) ||
211                         HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
212                         IREmit->GetOpHeader(IROp->Args[0]))) {
213                             IROp->Op = IR::OP_LOADMEM;
214                             Changed = true;
215                         }
216                     }
217                     if (IROp->Op == IR::OP_STOREMEMTSO && StackStatus == STACK) {
218                         if (HitsRegister(FEXCore::X86State::REG_RBP, IREmit,
219                         IREmit->GetOpHeader(IROp->Args[1])) ||
220                         HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
221                         IREmit->GetOpHeader(IROp->Args[1]))) {
222                             IROp->Op = IR::OP_STOREMEM;
223                             Changed = true;
224                         }
225                     }
226                 }
227
228                 CurrentBlock->Visited = true;
229                 CurrentBlock->State = StackStatus;
230             }
231
232             return Changed;
233         }
234
235         std::unique_ptr<Pass> CreateStackAccessTSORemovalPass() {
236             return std::make_unique<StackAccessTSORemovalPass>();
237         }
238     }  // namespace FEXCore::IR
```