DOI: 10.25559/SITITO.020.202401.139-148
УДК 004.4'422

Оригинальная статья

Оптимизация инструкций широкого доступа в память в архитектуре AArch64

В. В. Черноног[1*], Э. А. Гаджиев[2], А. Д. Добров[1]
[1] ООО «Техкомпания Хуавей», г. Москва, Российская Федерация
Адрес: 121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2
[*] norrilsk@gmail.com
[2] ООО «Коулмэн Сервисиз», г. Москва, Российская Федерация
Адрес: 115054, Российская Федерация, г. Москва, ул. Щипок, д. 5/7, стр. 2

Аннотация
Процессоры с архитектурой ARM все чаще применяются в облачных и высокопроизводительных вычислениях, для которых важно использовать аппаратные ресурсы наиболее эффективным образом, в связи с чем возрастает роль компиляторов и оптимизаций кода. На производительность выполнения программы существенное влияние оказывают операции с памятью. В наборе команд Aarch64 имеются инструкции, позволяющие выполнять доступ в память к нескольким последовательным значениям (LDP, LDPSW, STP), они используются в локальных оптимизациях компиляторов для замены двух последовательных операций доступа в память. В этой работе продемонстрировано, что такая замена может быть неэффективной, если целью оптимизации не является уменьшение объема кода. Предложен алгоритм оптимизации, выполняющей поиск и замену двумя операциями чтения тех инструкций LDP (LDPSW), часть данных для которых может быть получена из очереди на запись в память. Оценка производительности показывает эффективность оптимизации, значительное ускорение получено на тестах rapidjson (5% при запуске на одном ядре и 3% при запуске на всех ядрах), tpcc (16% на одном ядре) и tpch (2% на всех ядрах).

Ключевые слова: компилятор, оптимизация, память, доступ в память, инструкция чтения, процессор, система команд, Aarch64, ARM, производительность

Конфликт интересов: авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Черноног В. В., Гаджиев Э. А., Добров А. Д. Оптимизация инструкций широкого доступа в память в архитектуре AArch64 // Современные информационные технологии и ИТ-образование. 2024. Т. 20, № 1. С. 139-148. https://doi.org/10.25559/SITITO.020.202401.139-148

Original article

Compiler Optimization of Wide Memory Access for Aarch64

V. V. Chernonog[a*], E. A. Gadzhiev[b], A. D. Dobrov[a]
[a] Huawei Technologies Co., Ltd, Moscow, Russian Federation
Address: 17 Krylatskaya St., building 2, Moscow 121614, Russian Federation
[*] norrilsk@gmail.com
[b] Coleman group, Moscow, Russian Federation
Address: 5/7 Schipok St., building 2, Moscow 115054, Russian Federation

Abstract
The popularity of ARM-based processors in HPC and cloud computing is gradually increasing and the development of compilers and code optimizations is important to achieve better use of hardware resources. It is well known that memory operations have a significant impact on performance. There are load and store instructions in Aarch64 ISA (LDP, LDPSW, STP) that can be used to access a wide memory range. Compilers use these instructions in architecture-dependent peephole optimizations to replace sequence of load/store instructions. In this paper, the pros and cons of such optimization are considered, some cases are taken into account and analyzed by simulation to identify a potential performance improvement opportunity. Next, compiler optimization is proposed to find and replace with two memory loads those LDP (LDPSW) instructions part of the data of which can be obtained by the store-to-load forwarding hardware mechanism. The performance evaluation shows that the proposed optimization improves performance for applications RapidJSON (by 5% on single-core run and 3% on all-core run), tpcc (16% on single-core run) and tpch (2% on all-core run).

Keywords: compiler, optimization, memory, memory access, memory disambiguation, store queue, load instruction, processor, instruction set, Aarch64, ARM, performance

Conflict of interests: The authors declare no conflict of interest.

For citation: Chernonog V.V., Gadzhiev E.A., Dobrov A.D. Compiler Optimization of Wide Memory Access for Aarch64. *Modern Information Technologies and IT-Education.* 2024;20(1):139-148. https://doi.org/10.25559/SITITO.020.202401.139-148

Введение

Проблема оптимизации программного кода остается актуальной на сегодня и особенно критична для высокопроизводительных и облачных вычислений [1]. Под целью оптимизации может подразумеваться не только уменьшение времени исполнения программы, но и повышение энергоэффективности вычислений, а также сокращение объема кода. За время существования вычислительных машин инженеры придумали множество методов улучшения производительности как посредством изменения аппаратной составляющей процессора, так и посредством модификации программ.

Программная оптимизация может быть выполнена вручную, для чего нужно обладать хорошими знаниями архитектуры целевого процессора и навыками профилирования и низкоуровневого программирования. Существует большое количество способов: выравнивание структур данных, программная предвыборка кода и данных (prefetch), встраивание функций, векторизация и прочие модификации [2-8]. Зачастую эту работу лучше переложить на оптимизирующий компилятор, который может оперировать кодом на более низком уровне вплоть до ассемблерных инструкций [7, 9, 10]. В компилятор разработчиками закладываются особенности существующей архитектуры, служащие основой архитектурно-зависимых оптимизаций. Они, в совокупности с архитектурно-независимыми трансформациями, позволяют повысить эффективность использования аппаратных ресурсов. Более того, исполнение конкретно взятой программы можно сделать еще быстрее посредством детальной конфигурации опций компилятора, комбинируя их и подбирая значения параметров (tuning), которыми оперируют оптимизации [11, 12].

Под изменением аппаратной составляющей имеется в виду усложнение микроархитектуры процессора посредством добавления блоков, позволяющих выполнять код спекулятивно, разрешать зависимости, возникающие по ходу исполнения и замедляющие конвейер, обеспечивать быстрый доступ в память (кэш-память), осуществлять операции над векторными данными и многое другое[1]. Были разработаны различные подходы к проектированию архитектуры процессора и его набора команд, но на сегодня широко применяются подходы CISC и RISC [13-15]. Если говорить о вычислителях, применяющихся в серверах и суперкомпьютерах, то на протяжении долгого времени в этой нише лидировали процессоры со сложной CISC-архитектурой, в то время как RISC-процессоры, являясь более энергоэффективными и имея простую систему команд, использовались в мобильных устройствах. В последнее время ситуация на рынке изменилась, появились серверные и суперкомпьютерные процессоры с RISC-архитектурой, которые сохраняют свои свойства энергоэффективности и к тому же близко подбираются к лидерам по производительности[2] [16-20]. Особенно конкурентоспособными сейчас являются вычислители с архитектурой Aarch64, которая будет рассмотрена в данной работе.

Инструкции чтения памяти и зависимости по данным

В наборе команд Aarch64, помимо простых инструкций загрузки одного регистра (LDRB, LDRH, LDR), есть инструкции загрузки пары регистров общего назначения или пары векторных регистров (LDP, LDPSW), которые порождают запрос доступа к широкой области памяти (шире, чем размер регистра). В основном эти инструкции используются для генерации стекового фрейма, однако этим их использование не ограничено. Компилятором может применяться локальная оптимизация (peephole), в результате которой несколько инструкций чтения или записи в память будут объединены в одну инструкцию[3]. Такие трансформации имеются, например, в популярных компиляторах GCC[4] и LLVM[5] (CLang). Замена позволяет сократить размер объектного кода, а микроархитектурная реализация исполнения инструкций широкого доступа в память может быть эффективнее исполнения двух раздельных инструкций.

Независимо от типа системы команд инструкции доступа к памяти могут сильно увеличивать время выполнения программы. При наличии зависимостей по данным исполнение кода приостанавливается до момента получения актуального значения из памяти. Одним из аппаратных механизмов разрешения зависимостей по данным является ранний доступ через буфер очереди на запись (store-to-load forwarding). Его принцип состоит в том, что актуальные данные можно получить напрямую из очереди на запись в память, не ожидая, пока запись будет выполнена[6].

Если в структуре конвейера имеется два и более функциональных блока для операций чтения и записи, то использование инструкции LDP уменьшает число задействованных блоков. Но возможна следующая ситуация[7]:

1. Выполняется инструкция записи в память (STR, STP) по некоторому адресу.
2. Операция записи помещается в буфер записи.
3. Спустя несколько операций выполняется инструкция чтения пары значений из памяти (LDP), причем запрашиваемый диапазон наполовину пересекается с диапазоном предшествующей операции записи.

[1] Hennessy J. L., Patterson D. A. Computer Architecture. Fifth Ed: A Quantitative Approach (5th. ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[2] Кузьминский М. Б. Современные серверные ARM-процессоры для суперЭВМ: A64FX и другие. Начальные данные тестов производительности // Программные системы: теория и приложения. 2022. Т. 13, № 1(52). С. 63-129. https://doi.org/10.25209/2079-3316-2022-13-1-63

[3] Gourdin L. PhD Student session: formally verified postpass scheduling with peephole optimization for AArch64 [Электронный ресурс] // Bienvenue Aux Journées Afadl. 2021.

[4] Локальные оптимизации Load/Store инструкций [Электронный ресурс] // Репозиторий GCC, 2024. URL: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/config/aarch64/aarch64-ldpstp.md;h=1ee7c73ff0c8de926f9e82731904354dd642377a;hb=refs/heads/releases/gcc-13 (дата обращения 02.02.2024).

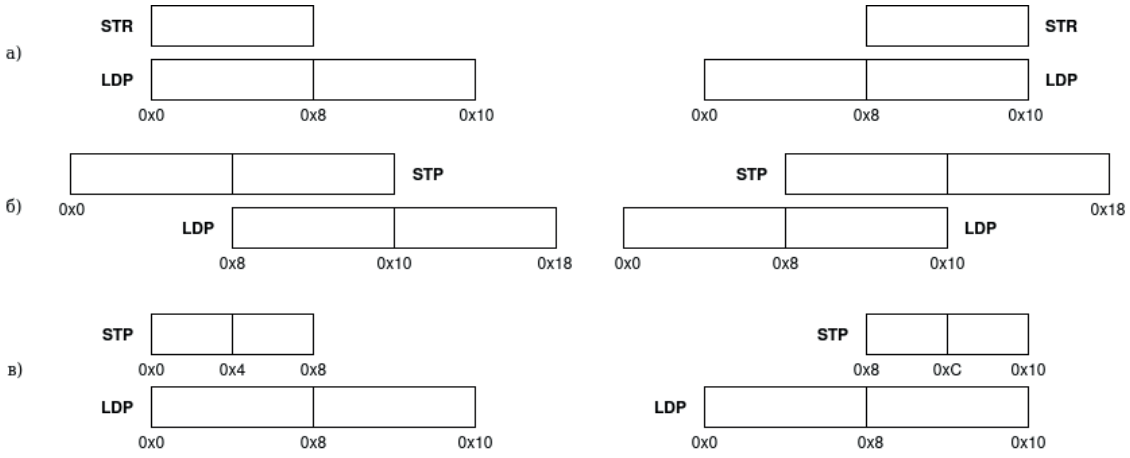
[5] Локальные оптимизации Load/Store инструкций [Электронный ресурс] // Репозиторий LLVM, 2024. URL: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/AArch64/AArch64LoadStoreOptimizer.cpp (дата обращения 02.02.2024).

[6] Shen J. P., Lipasti M. H. Modern Processor Design: Fundamentals of Superscalar Processors. Waveland Press, 2013. 642 p.

[7] Там же.

4. Актуальные данные не могут быть получены из очереди на запись, так как для этого запрашиваемый диапазон должен полностью содержаться в одной из записей, находящихся в очереди.
5. Инструкция LDP помещается в очередь на чтение до окончания необходимой записи в память.

Отметим, что время выполнения операций между рассматриваемыми двумя инструкциями записи и чтения не должно превышать времени завершения записи в память. Все возможные случаи пересечения диапазонов адресов показаны на рисунке 1 (указанные адреса являются условными):
– на рисунке 1а - диапазон памяти, на который ссылается инструкция STR, является половиной диапазона памяти инструкции чтения LDP, идущей далее;
– на рисунке 1б - диапазоны памяти инструкций STP и LDP пересекаются ровно наполовину;
– на рисунке 1в - инструкция STP содержит пару регистров меньшего размера (32 бита), и диапазон памяти, как в первом случае, является половиной диапазона инструкции LDP.



Р и с. 1. Пересечения диапазонов адресуемой памяти
F i g. 1. Memory range intersections

Источник: здесь и далее в статье все таблицы и рисунки составлены авторами.
Source: Hereinafter in this article all tables and figures were made by the authors.

Для некоторых процессоров архитектуры ARM инструкции доступа к широкой области памяти не могут быть выставлены на исполнение за один такт, и явное разделение таких инструкций на несколько позволяет обойти это ограничение. Также появляется возможность загрузить свободные функциональные блоки одной или несколькими арифметическими операциями в один такт вместе с инструкцией доступа в память [16], [21].

Таким образом, значение одного из регистров может быть получено раньше, и если регистр используется далее по коду, то быстрое чтение значения из буфера записи позволит избежать лишних задержек конвейера. В этой работе предлагается решение проблемы зависимости по данным посредством оптимизации, осуществляющей разделение тех инструкций чтения, для которых половина адресуемой области может быть получена из очереди на запись, где содержится предшествующая операция записи в память.

Проблема выравнивания адресов памяти

Одной из причин, по которой операции доступа в память вызывают длительные задержки исполнения, является невыровненный адрес, по которому осуществляется доступ. Если такие запросы могут быть обработаны процессором, то будет выполнен доступ к нескольким ячейкам памяти, в которые входит запрашиваемая область, и тем самым увеличивается задержка.

В результате локальной оптимизации, заменяющей несколько инструкций чтения или записи на одну инструкцию широкого доступа, может быть потеряно свойство выравненности адреса, что и было обнаружено в случае с инструкцией LDP: операция чтения пары значений (16 байт) по адресу, не выровненному на 16 байт, выполняется дольше, чем две операции чтения 8 байт данных. В силу того, что определение свойств адресуемой памяти на этапе компиляции требует сложного анализа кода, эта проблема оставлена для дальнейших исследований.

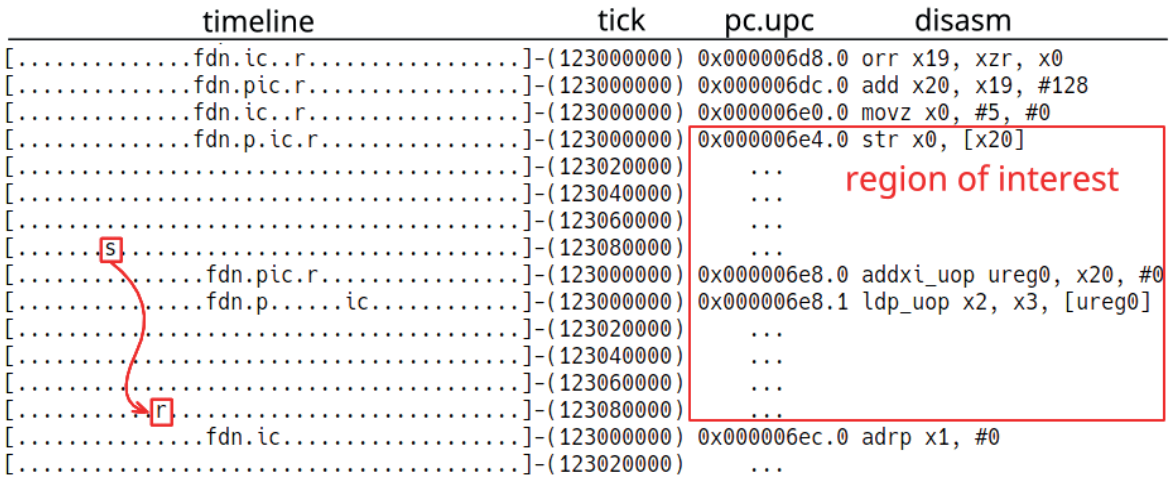
Предварительная оценка в симуляторе

Моделирование ситуации позволяет проанализировать поведение микроархитектуры и выполнить предварительную оценку производительности. Для этих целей использован симулятор gem5, который содержит детальную модель суперскалярного процессора на основе Alpha 21264, а также модель архитектуры ARM [22, 23]. Конфигурация симулируемой системы приведена в Таблице 1. Отметим, что моделирование осуществляется в режиме эмуляции системных вызовов (system emulation), в котором отсутствует исполнение кода ядра операционной системы.

Т а б л и ц а 1. **Конфигурация модели процессора**
T a b l e 1. **Configuration of the simulated CPU**

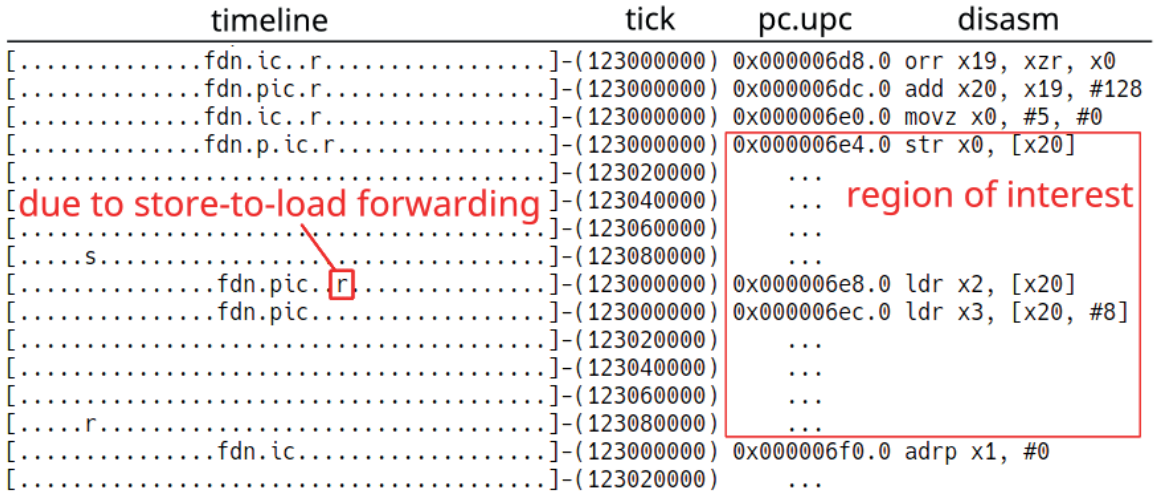
Архитектура	ARMv8.2-A
Модель процессора	ArmO3CPU
Кэш-память L1	64 кБ кэш данных, 64 кБ кэш инструкций
Кэш-память L2	512 кБ, общий
Ширина этапов конвейера	4 для всех этапов
Количество LSU блоков	2
Оперативная память	DDR4 2400 МГц 512 МБ

На рисунке 2 приводится код, содержащий последовательность инструкций STR и LDP с одинаковым базовым регистром. Визуализация перемещения инструкций в конвейере выполнена с помощью логгирования работы конвейера (*O3PipeView*) и специальной утилиты в составе gem5 (*o3-pipeviewer.py*). В левой части находится временная шкала (каждая точка соответствует такту процессора), где указаны стадии конвейера (f - fetch, d - decode, n - rename, p - dispatch, i - issue, c - complete, r - retire, s - store-complete), в правой части - инструкции. Согласно инструкции LDP процессором выполняется чтение 16 байт данных, из которых младшие 8 байт обновляются предшествующей инструкцией STR. Возникает конфликт типа “чтение после записи”, из-за чего инструкция LDP, которая разбита на микрооперации, переходит на этап retire только спустя 4 такта после завершения записи в память.



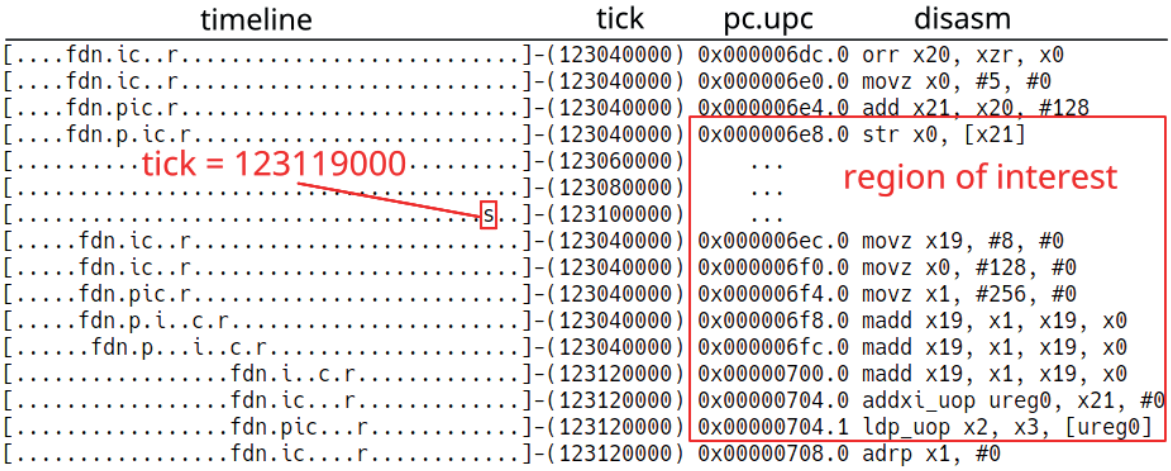
Р и с. 2. Моделирование исполнения инструкции LDP после записи
F i g. 2. Simulation an execution of LDP after store

На рисунке 3 проиллюстрированы результаты исполнения программы, в которой содержатся две инструкции чтения памяти. Исполнение первой инструкции загрузки регистра завершается быстро благодаря аппаратному механизму разрешения зависимостей при спекулятивном выполнении операций доступа в память.



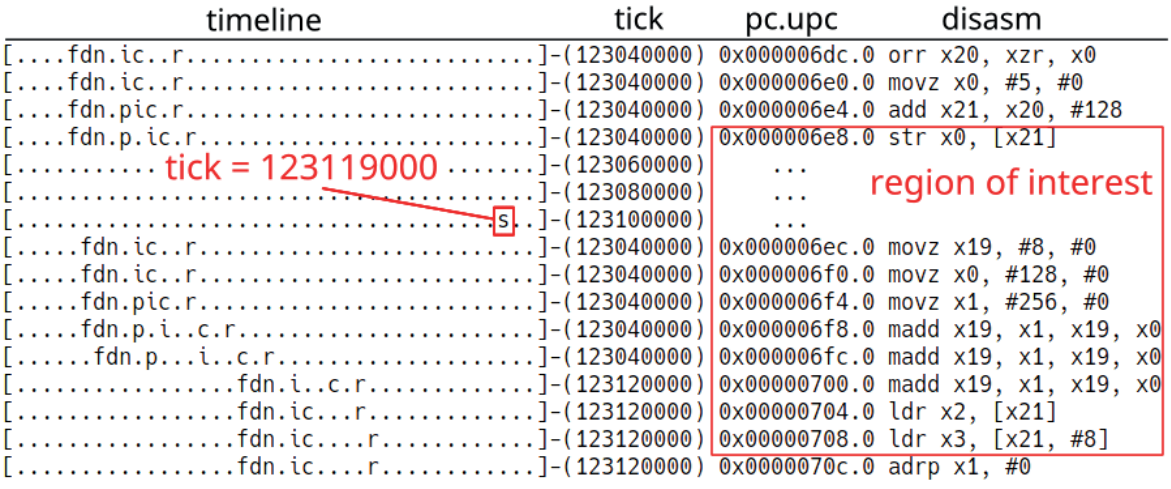
Р и с. 3. Моделирование исполнения двух инструкций чтения памяти
F i g. 3. Simulation an execution of two memory loads after store

Как было отмечено выше, между рассматриваемыми инструкциями чтения и записи процессором могут выполняться прочие операции, и к моменту начала исполнения операции чтения запись в память полностью завершится. Моделирование такой ситуации для двух ранее рассмотренных случаев продемонстрировано на рисунках 4-5. В обоих случаях на чтение пары значений тратится одинаковое время, так как в очереди на запись уже нет данных, которые можно взять для первой инструкции чтения. Следовательно, разделение инструкции LDP не всегда целесообразно и требует анализа кода. В общем случае максимальное число инструкций между двумя конфликтующими операциями доступа в память, которое далее будем называть дистанцией, зависит от конфигурации конкретного процессора и времени выполнения инструкций различного типа, поэтому предлагается определять это значение эмпирическим путем.



Р и с. 4. Моделирование исполнения арифметических операций перед инструкцией LDP

F i g. 4. Simulation an execution of arithmetic instructions before LDP



Р и с. 5. Моделирование исполнения арифметических операций перед двумя инструкциями чтения

F i g. 5. Simulation an execution of arithmetic instructions before two memory loads

Алгоритм поиска конфликтующих инструкций

Для поиска инструкции чтения LDP, которую нужно разделить на две инструкции LDR, предлагается следующий алгоритм оптимизации. В листинге 1 приведен псевдокод главной процедуры, в качестве единицы оптимизации рассматривается функция. Для каждой инструкции LDP выполняется поиск кандидатов (*find_canditates*) и обратный проход графа потока исполнения на некоторую дистанцию – число инструкций (*distance*) – с целью поиска ранее помеченных кандидатов. Дистанция является параметром и может задаваться через соответствующую опцию компилятора. Переменная *store_candidates* – контейнер-множество, содержащий инструкции записи, пересекающиеся по диапазону памяти с текущей инструкцией чтения. Последним выполняется разделение тех инструкций (*split_insn*), для которых найденные кандидаты находились на указанной дистанции.

```
procedure pass_split_ldp ():
 for basic_block bb in func:
  for instruction insn in bb:
   if insn is not LDP:
    continue
   find_candidates (insn)
   if not store_candidates.empty ()
     and find_by_bfs (insn, bb, insn, distance):
    ldp_to_split.insert (insn)
 for instruction insn in ldp_to_split:
  split_insn (insn)
```

Листинг 1. Псевдокод входной процедуры
Listing 1. Entry procedure pseudocode

Алгоритм поиска кандидатов (листинг 2) требует наличия схемы достигающих определений, для чего необходимо выполнить анализ потока данных. Функция *get_base_reg* возвращает базовый адресный регистр инструкции доступа к памяти, ее реализация зависит от используемого компилятора и от формата промежуточного представления кода на данном этапе. Функции *get_definitions* и *get_uses*, используя схему достигающих определений, возвращают список определений базового адресного регистра и список его использований, соответственно. Инструкция записи в память помещается в множество *store_candidates*, если она имеет тот же базовый регистр, что инструкция LDP, и если диапазон записываемой памяти является половиной диапазона памяти инструкции LDP или имеет пересечение, кратное размеру регистра (функция *ranges_overlap*).

```
procedure find_candidates (load_insn):
 store_candidates ← ∅
 ldp_base_reg ← get_base_reg (load_insn)
 defs ← get_definitions (ldp_base_reg)
 for def in defs:
  uses ← get_uses (def)
  for use in uses:
   insn ← use.insn   // instruction that contains the use
   insn_base_reg ← get_base_reg (insn)
   if insn is STORE instruction and insn_base_reg = ldp_base_reg
     and ranges_overlap(insn, load_insn):
    store_candidates.insert (insn)
```

Листинг 2. Псевдокод процедуры поиска инструкций записи
Listing 2. Pseudocode of store instruction search

На этапе обратного прохода графа потока исполнения в ширину, осуществляемого функцией *find_by_bfs* (листинг 3), выполняется поисковый цикл по инструкциям базового блока с рекурсией при переходе к предшествующим базовым блокам.

```
function find_by_bfs (ldp, bb, current_insn, distance):
 for i ← distance to 1:
  if store_candidates.contains (current_insn):
   return true
  if current_insn is head of bb:
   for each edge in bb.predecessors_edges:
    if find_by_bfs (ldp, edge.bb, edge.bb.last_insn, i - 1):
     return true
   return false
  else
   current_insn ← current_insn.previous_insn
 return false
```

Листинг 3. Псевдокод функции обратного поиска в ширину
Listing 3. Pseudocode of reverse breadth-first search

Псевдокод процедуры *split_insn* не приводится, в зависимости от формата промежуточного представления, ее реализация подразумевает либо генерацию новых инструкций, либо копирование представлений из существующей инструкции и их встраивание в код в качестве отдельных операций.

Пример разделения инструкции

Для оценки эффективности рассматриваемой оптимизации в компиляторе GCC версии 10.3 был реализован приведенный ранее алгоритм поиска и разделения инструкций чтения пары значений. В результате оптимизации производится разделение следующих RTL-представлений (Register Transfer Language):
1. Для инструкций чтения вида
parallel [(set (reg1) (mem (base_reg)))
  (set (reg2) (mem (plus (base_reg) (offset))))]
и инструкций чтения с пост-индексной операцией вида
parallel [(set (base_reg) (value))
(set (reg1) (mem (base_reg)))
(set (reg2) (mem (plus (base_reg) (offset))))]
генерируются 2 и 3 инструкции, соответственно, посредством копирования внутренних set-инструкций и их встраивания в код. При этом учитывается, что операция изменения значения базового регистра должна быть добавлена последней.
2. Для инструкций чтения вида
set (reg1) (mem (base_reg))
выполняется генерация новых инструкций, номер второго регистра в такой инструкции всегда будет следующим за указанным номером первого регистра, а смещение второй инструкции чтения равно половине размера данных, указанных в исходной инструкции.

Оценка производительности

Для тестирования использовался серверный процессор Kunpeng 920 с микроархитектурой Taishan v110, которая базируется на архитектуре ARMv8.2-A [24]. Особенностью данной модели является большое количество вычислительных ядер и большой объем кэш-памяти третьего уровня. Прочие характеристики процессора приведены в Таблице 2. За один такт процессор считывает из памяти 4 инструкции, в числе его функциональных блоков имеется два блока чтения/записи памяти (LSU).

Таблица 2. **Характеристики процессора Kunpeng 920**
Table 2. **Kunpeng 920 characteristics**

Кэш-память L1	64 кБ кэш данных, 64 кБ кэш инструкций
Кэш-память L2	512 кБ, общий для данных и инструкций
Кэш-память L3	2 x 64 МБ, разделяемый (1 МБ/ядро)
Оперативная память	DDR4 2933 МГц
Число физических ядер	2 x 64
Частота	3 ГГц

Оценка влияния оптимизации на время выполнения производилась на пакете тестовых программ CPUBench [25]. В его состав входят программы с целочисленными вычислениями и с вычислениями с плавающей точкой. Для процессора Kunpeng 920 минимальное время доступа в память составляет 4 такта (доступ в кэш-память L1), а так как он способен загружать из памяти 4 инструкции за такт, то величина дистанции через опцию компилятора была установлена равной 16. Осуществлены два метода тестирования: в первом оценивается производительность процессора при выполнении каждого теста на одном ядре (конфигурации IntSingle, FloatSingle), во втором оценивается пропускная способность, когда каждый тест запускается на всех ядрах одновременно (конфигурации IntConcurrent, FloatConcurrent).

В Таблице 3 приведены результаты оценки производительности для конфигурации IntSingle, по которым видно, что наибольшее число инструкций было найдено в программах *tpcc* и *tpch*. Показатель производительности определяется отношением измеренного времени выполнения к времени выполнения на некотором эталонном процессоре, умноженным на число копий приложения, запущенных на процессоре [26]. Время выполнения сократилось для тестов *tpcc* и *rapidjson*, для остальных тестов оно сохранилось с небольшим отличием. На рисунке 6 проиллюстрированы диаграммы времени выполнения каждого теста.

Таблица 3. **Результаты оценки применения оптимизации на конфигурации IntSingle**
Table 3. **Performance evaluation on IntSingle benchmark configuration**

Название теста	Количество разделенных инструкций LDP	Показатель производительности выполнения с базовыми опциями	Показатель производительности выполнения с опцией разделения LDP	Ускорение
tpcc	17063	1,20	1,39	1,16
rapidjson	110	1,17	1,22	1,05
x264	51	2,26	2,29	1,01
velvet	25	1,51	1,54	1,01
openssl	41	3,01	3,01	1,00
tpch	17063	1,41	1,40	1,00
gzip	3	1,89	1,89	1,00
python	208	1,25	1,24	0,99
xz	4	1,76	1,75	0,99
gcc	16162	1,31	1,31	0,99

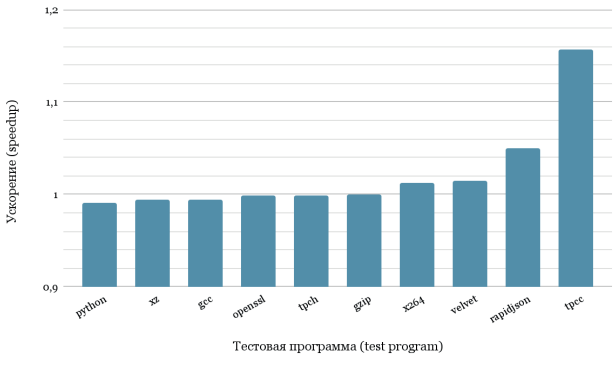


Рис. 6. Ускорение выполнения бенчмарка CPUBench IntSingle
Fig. 6. Speedup of CPUBench IntSingle

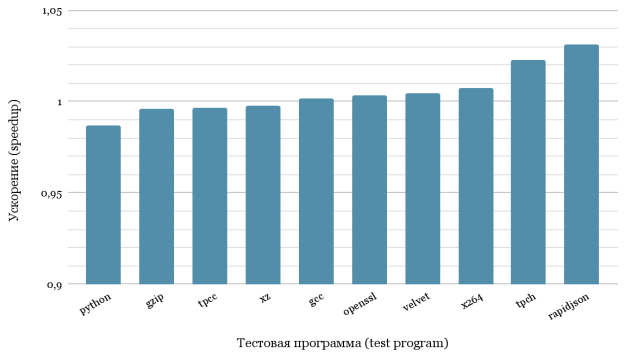
В Таблице 4 представлены показатели производительности, измеренные на конфигурации IntConcurrent. На рисунке 7 проиллюстрированы диаграммы времени выполнения каждой тестовой программы. На этой конфигурации наибольшее ускорение наблюдается для программ *rapidjson* и *tpch*.

Результаты оценки на конфигурациях FloatSingle и FloatConcurrent не показали улучшения производительности, время выполнения тестовых программ не изменилось.

Таблица 4. **Результаты оценки применения оптимизации на конфигурации IntConcurrent**
Table 4. **Performance evaluation on IntConcurrent configuration**

Название теста	Показатель производительности выполнения с базовыми опциями	Показатель производительности выполнения с опцией разделения LDP	Ускорение
rapidjson	123,36	127,22	1,03
tpch	134,03	137,09	1,02
x264	268,59	270,58	1,01
gzip	237,13	236,21	1,00
tpcc	166,97	166,40	1,00
xz	156,22	155,89	1,00

Название теста	Показатель производительности выполнения с базовыми опциями	Показатель производительности выполнения с опцией разделения LDP	Ускорение
gcc	121,48	121,67	1,00
openssl	379,95	381,26	1,00
velvet	90,40	90,80	1,00
python	154,74	152,68	0,99



Р и с. 7. Ускорение выполнения бенчмарка CPUBench IntConcurrent

F i g. 7. Speedup of CPUBench IntConcurrent

Заключение

Результаты оценки производительности показали эффективность оптимизации для некоторых приложений, согласно собранной статистике в них было найдено большое количество конфликтующих операций записи и чтения памяти. Разделение инструкции LDP (LDPSW) позволяет ускорить спекулятивное исполнение, когда прочитанное значение может быть использовано сразу далее по коду. Возможность вручную задавать значение дистанции позволяет настроить оптимизацию под конкретный процессор. Предложенную трансформацию необходимо выполнять после распределения регистров (register allocation) и локальных трансформаций (peephole optimizations), чтобы исключить обратную замену сгенерированных инструкций.

В качестве дальнейшего исследования можно рассмотреть операции доступа в память с векторными регистрами, операции доступа в память по адресам, не выровненным на размер запрашиваемых данных, а также выполнить оценку эффективности оптимизации на прочих бенчмарках и процессорах.

References

[1] Schmitt N., Bucek J., Beckett J., Cragin A., Lange K.-D., Kounev S. Performance, Power, and Energy-Efficiency Impact Analysis of Compiler Optimizations on the SPEC CPU 2017 Benchmark Suite. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). Leicester, UK: IEEE Computer Society; 2020. p. 292-301. https://doi.org/10.1109/UCC48980.2020.00047

[2] Lebeck A.R., Wood D.A. Cache profiling and the SPEC benchmarks: a case study. *Computer*. 1994;27(10):15-26. https://doi.org/10.1109/2.318580

[3] Jain R., Cheng S., Kalagi V., Sanghavi V., Kaul S., Arunachalam M., Maeng K., Jog A., Sivasubramaniam A., Kandemir M.T., Das C.R. Optimizing CPU Performance for Recommendation Systems At-Scale. In: Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23). New York, NY, USA: Association for Computing Machinery; 2023. Article number: 77. https://doi.org/10.1145/3579371.3589112

[4] Kelefouras V., Djemame K. A methodology for efficient code optimizations and memory management. In: Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18). New York, NY, USA: Association for Computing Machinery; 2018. p. 105-112. https://doi.org/10.1145/3203217.3203274

[5] Mitra G., Johnston B., Rendell A.P., McCreath E., Zhou J. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. Cambridge, MA, USA: IEEE Computer Society; 2013. p. 1107-1116. https://doi.org/10.1109/IPDPSW.2013.207

[6] Chhugani J., Nguyen A.D., Lee V.W., Macy W., Hagog M., Chen Y.-K., Baransi A., Kumar S., Dubey P. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment*. 2008;1(2):1313-1324. https://doi.org/10.14778/1454159.1454171

[7] Veras R., Popovici D.T., Low T.M., Franchetti F. Compilers, hands-off my hands-on optimizations. In: Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (WPMVP '16). New York, NY, USA: Association for Computing Machinery; 2016. Article number: 4. https://doi.org/10.1145/2870650.2870654

[8] Simunic T., Benini L., De Micheli G., Hans M. Source code optimization and profiling of energy consumption in embedded systems. In: Proceedings 13th International Symposium on System Synthesis. Madrid, Spain: IEEE Computer Society; 2000. p. 193-198. https://doi.org/10.1109/ISSS.2000.874049

[9] Debray S.K., Evans W., Muth R. De Sutter B. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems.* 2000;22(2):378-415. https://doi.org/10.1145/349214.349233

[10] Tan J., Jiao S., Chabbi M., Liu X. What every scientific programmer should know about compiler optimizations? In: Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20). New York, NY, USA: IEEE Computer Society; 2020. Article number: 42. https://doi.org/10.1145/3392717.3392754

[11] Plotnikov D., Melnik D., Vardanyan M., Buchatskiy R., Zhuykov R., Lee J.-H. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. *Procedia Computer Science*. 2013;18:1312-1321. https://doi.org/10.1016/j.procs.2013.05.298

[12] Pan Z., Eigenmann R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: International Symposium on Code Generation and Optimization (CGO'06). New York, NY, USA: IEEE Computer Society; 2006. p. 12. https://doi.org/10.1109/CGO.2006.38
[13] George A. An overview of RISC vs. CISC. In: Proceedings The Twenty-Second Southeastern Symposium on System Theory. Cookeville, TN, USA: IEEE Computer Society; 1990. p. 436-438. https://doi.org/10.1109/SSST.1990.138185
[14] Jamil T. RISC versus CISC. *IEEE Potentials.* 1995;14(3):13-16. https://doi.org/10.1109/45.464688
[15] Blem E., Menon J., Sankaralingam K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). Shenzhen, China: IEEE Computer Society; 2013. p. 1-12. https://doi.org/10.1109/HPCA.2013.6522302
[16] Wiggers T. Energy-Efficient ARM64 Cluster with Cryptanalytic Applications. In: Lange T., Dunkelman O. (eds.) Progress in Cryptology – LATINCRYPT 2017. LATINCRYPT 2017. *Lecture Notes in Computer Science.* Vol. 11368. Cham: Springer; 2019. p. 175-188. https://doi.org/10.1007/978-3-030-25283-0_10
[17] Yokoyama D., Schulze B., Borges F., et al. The survey on ARM processors for HPC. *The Journal of Supercomputing.* 2019;75:7003-7036. https://doi.org/10.1007/s11227-019-02911-9
[18] Jiang Q., Lee Y.C., Zomaya A.Y. The Power of ARM64 in Public Clouds. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). Melbourne, VIC, Australia: IEEE Computer Society; 2020. p. 459-468. https://doi.org/10.1109/CCGrid49817.2020.00-47
[19] Afanasyev I., Lichmanov D. Evaluating the Performance of Kunpeng 920 Processors on Modern HPC Applications. In: Malyshkin V. (ed.) Parallel Computing Technologies. PaCT 2021. *Lecture Notes in Computer Science.* Vol. 12942. Cham: Springer; 2021. p. 301-321. https://doi.org/10.1007/978-3-030-86359-3_23
[20] McIntosh-Smith S., Price J., Deakin T., Poenaru A. A performance analysis of the first generation of HPC-optimized Arm processors. *Concurrency and Computation: Practice and Experience*. 2019;31(16):e5110. https://doi.org/10.1002/cpe.5110
[21] Park J., Kwon Y., Park Y., Jeon D. Microarchitecture-Aware Code Generation for Deep Learning on Single-ISA Heterogeneous Multi-Core Mobile Processors. *IEEE Access*. 2019;7:52371-52378. https://doi.org/10.1109/ACCESS.2019.2910559
[22] Binkert N., et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*. 2011;39(2):1-7. https://doi.org/10.1145/2024716.2024718
[23] Lowe-Power J., et al. The gem5 simulator: Version 20.0+. *arXiv:*2007.03152. 2020. https://doi.org/10.48550/arXiv.2007.03152
[24] Xia J., Cheng C., Zhou X., Hu Y., Chun P. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro*. 2021;41(5):67-75. https://doi.org/10.1109/MM.2021.3085578
[25] Lu H.T, Ren X., Zhong W.J., et al. CPUBench: An open general computing CPU performance benchmark tool[J]. *Microelectronics & Computer*. 2023;40(5):75-83. https://doi.org/10.19304/J.ISSN1000-7180.2022.0469
[26] Bucek J., Lange K.-D., v. Kistowski J. SPEC CPU2017: Next-Generation Compute Benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). New York, NY, USA: Association for Computing Machinery; 2018. p. 41-42. https://doi.org/10.1145/3185768.3185771

Поступила 07.12.2023; одобрена после рецензирования 02.02.2024; принята к публикации 11.03.2024.
Submitted 07.12.2023; approved after reviewing 02.02.2024; accepted for publication 11.03.2024.

Об авторах:

Черноног Вячеслав Викторович, сотрудник, ООО «Техкомпания Хуавей» (121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2), **ORCID: https://orcid.org/0009-0007-8407-1082,** norrilsk@gmail.com
Гаджиев Эмин Арифович, сотрудник, ООО «Коулмэн Сервисиз» (115054, Российская Федерация, г. Москва, ул. Щипок, 5/7 стр. 2), **ORCID: https://orcid.org/0009-0004-1702-9376,** e.gadzhiev.mhk@gmail.com
Добров Андрей Дмитриевич, сотрудник, ООО «Техкомпания Хуавей» (121614, Российская Федерация, г. Москва, ул. Крылатская, д. 17, к. 2), кандидат технических наук, **ORCID: https://orcid.org/0009-0007-5074-7873,** adobrov1954@gmail.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Viacheslav V. Chernonog, Researcher, Huawei Technologies Co. Ltd. (17 Krylatskaya St., building 2, Moscow 121614, Russian Federation), **ORCID: https://orcid.org/0009-0007-8407-1082,** norrilsk@gmail.com
Emin A. Gadzhiev, Researcher, Coleman group (5/7 Schipok St., building 2, Moscow 115054, Russian Federation), **ORCID: https://orcid.org/0009-0004-1702-9376,** e.gadzhiev.mhk@gmail.com
Andrey D. Dobrov, Researcher, Huawei Technologies Co. Ltd. (17 Krylatskaya St., building 2, Moscow 121614, Russian Federation), Cand. Sci. (Eng.), **ORCID: https://orcid.org/0009-0007-5074-7873,** adobrov1954@gmail.com

All authors have read and approved the final manuscript.