DOI: 10.15514/ISPRAS-2025-37(6)-35

Аппаратное ускорение модуля MMU при полносистемной эмуляции aarch64 на x86-64 в эмуляторе Qemu

Д.Н. Полетаев, ORCID: 0009-0005-8872-2802 <dmitry.poletaev@ispras.ru>
П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
Г.Н. Тейс, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>
М.А. Костин, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Новгородский государственный университет им. Ярослава Мудрого,
Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.

Аннотация. В работе рассматривается оптимизация, позволяющая переложить часть вычислений, связанных с трансляцией гостевых виртуальных адресов при полносистемной эмуляции с программного MMU на MMU хостовой системы. Описан вариант применения оптимизации к эмулятору Qemu. Выполнена оценка прироста производительности от использования оптимизации, предложен вариант дальнейшего развития подхода.

Ключевые слова: виртуальные машины; полносистемная эмуляция; аппаратный MMU; ускорение эмуляции.

Для цитирования: Полетаев Д.Н., Довгалюк П.М., Тейс Г.Н., Костин М.А. Аппаратное ускорение модуля MMU при полносистемной эмуляции aarch64 на x86-64 в эмуляторе Qemu. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 45–58. DOI: 10.15514/ISPRAS–2025–37(6)–35.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022, https://rscf.ru/project/24-11-20022/.

Hardware Acceleration of Qemu MMU for aarch64 on x86-64 Full System Emulation

D.N. Poletaev, ORCID: 0009-0005-8872-2802 <dmitry.poletaev@ispras.ru>
P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
G.N. Teys, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>
M.A. Kostin, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Yaroslav-the-Wise Novgorod State University,
41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.

Abstract. Full system cross-ISA emulation is widely used nowdays, but is known for being slow. Major contribution to the slowdown is made by software MMU doing guest virtual addresses translation. In article we look at optimization which allows to move part of such address translation work to the hardware MMU of the host system. For this goal, extra view to the whole guest virtual address space is added to the address space of the emulator process, using mmap system call. After mapping is done there is opportunity to use fixed offset correction to guest virtual address in the translated binary code in place of dynamic search of needed offset in software TLB. Additional view of guest virtual address space maintained coherent with guest page tables. Such approach allows to use less host instructions per each guest memory instruction, which lead to notable emulation acceleration, considering the large quantity of memory instructions in the guest execution flow. Measurments show speed up as large as 271% for benchmark tests and up to 217% for the real-world program. Ideas are proposed for overcoming some limitations of described approach.

Keywords: virtual machines; full system emulation; hardware MMU; emulation speed up.

For citation: Poletaev D.N., Dovgaluk P.M, Teys G.N., Kostin M.A. Hardware acceleration of Qemu MMU for aarch64 on x86-64 full system emulation. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 45-58 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-35.

Acknowledgements. This work is supported by Russian Science Foundation under grant number 24-11-20022, https://rscf.ru/project/24-11-20022/.

1. Введение

Кросс-архитектурная полносистемная эмуляция активно используется в сферах разработки компьютерной аппаратуры и программного обеспечения, цифровой безопасности, моделировании систем и пр. Одним из основных неудобств, а иногда и ограничений, связанных с применением полносистемной эмуляции для этих целей, считается низкая производительность программных эмуляторов (замедление работы эмулируемого ПО в десятки раз – не редкость). Большой вклад в замедление вносит эмуляция работы виртуальной памяти, которая является неотъемлемой частью многих компьютерных архитектур, включая aarch64. По оценке из работы [1], накладные расходы на эмуляцию виртуальной памяти в полносистемных эмуляторах на основе двоичной трансляции могут достигать ~40% от всего замедления, вносимого эмуляцией. В работе рассматривается оптимизация, позволяющая использовать аппаратуру MMU хостовой (на которой работает эмулятор) системы для эмуляции работы виртуальной памяти гостевой (которая запущена в эмуляторе) системы. Для практической апробации подхода выбран эмулятор Qemu. Результаты выполненных замеров работы с применением оптимизации показывают значительное увеличение производительности эмулятора в определенных режимах работы гостевой системы.

2. Обзор предлагаемых решений

2.1 Виртуальная память

Концепция виртуальных адресов позволяет изолировать различные процессы в системе друг от друга, а также эффективнее использовать доступную физическую память. В современных ОС (и платформах, на которых они работают) виртуальная память является страничной - все адресное пространство разбивается на равные непрерывные блоки памяти, страницы. Виртуальные страницы отображаются на страницы физической памяти, причем несколько виртуальных страниц могут быть отображены на одну физическую страницу, но не наоборот. Для описания того, каким физическим адресам соответствуют виртуальные адреса, существуют системные структуры памяти, хранящиеся в памяти, называемые страничными таблицами. Таблицы настраиваются системными программами (операционной системой), согласно его стратегии работы с памятью.

При каждом обращении процессора к памяти (в том числе для выбора инструкции), виртуальный адрес преобразуется специальным аппаратным блоком – устройством управления памятью (Memory Management Unit, MMU), который просматривает таблицы трансляций, в физический, который уже и используется, непосредственно, для доступа. Для кеширования результатов трансляций в MMU присутствует кеш трансляций – буфер ассоциативной трансляции (Translation Lookaside Buffer, TLB). Буфер TLB позволяет значительно снизить нагрузку на ОЗУ от работы механизма виртуальной памяти: если трансляция уже есть в TLB, то используется она, а полный поиск в таблицах не выполняется.

Таблицы трансляций динамично меняются по мере работы программы (память выделяется, процессы переключаются), и для обеспечения синхронного состояния таблиц и записей в TLB системное программное обеспечение (ПО) удаляет неактуальные записи с помощью специальных процессорных команд. Тогда при новом обращении по измененному адресу запись в TLB не найдется, и будет выполнен полный обход таблиц, а запись с актуальной трансляцией помещена в TLB.

В ситуациях, когда в таблицах отсутствует информация, как нужно транслировать определенный виртуальный адрес в физический адрес, или параметры доступа к странице памяти нарушаются (например, запись в страницу только для чтения или доступ из пользовательского процесса к памяти привилегированного процесса), в процессоре происходит исключение, и управление передается программе-обработчику исключения.

2.2 Эмуляция виртуальной памяти в Qemu

Qemu – полносистемный кроссплатформенный эмулятор с открытым исходным кодом [2]. Он поддерживает многие популярные компьютерные архитектуры, как в качестве гостевых систем, так и в качестве хостовых.

Работа эмулятора основана на динамической двоичной трансляции кода гостевой архитектуры в код хостовой архитектуры. Но чтобы избежать поддержки большого количества трансляторов для каждого сочетания исходной и целевой архитектуры, и для эффективного выполнения некоторых оптимизаций над кодом, трансляция кода выполняется с дополнительным шагом в виде промежуточного кода. Операции, которые сложно и/или неоправданно преобразовывать с помощью кодогенерации (редкие платформозависимые инструкции, меняющие состояние гостевой системы, трансляция виртуального адреса и прочие), интерпретируются с помощью функций на языке Си, а вызовы к этим функциям встраиваются непосредственно в сгенерированный двоичный код.

2.2.1 Программное управление памятью эмулятора QEMU

Для эмуляции виртуальной памяти гостевых архитектур в Qemu присутствует модуль программного управления памятью MMU. Модуль функционально представляет собой

модель буфера TLB, с набором оптимизаций для ускорения программной трансляции адресов, интерфейс для добавления трансляций в TLB, удаления трансляций, синхронизации транслированного кодогенератором кода с гостевым (в случае изменения гостевых страниц).

Модель TLB в Qemu платформонезависимая и универсальная, позволяет абстрагироваться от формата таблиц трансляции конкретной гостевой архитектуры – непосредственно обходом таблиц трансляции в гостевой системе занимается код эмуляции конкретной архитектуры.

2.2.2 Поиск трансляции в программном TLB эмулятора Qemu

Во время выполнения транслированного кода при обращении к памяти, как и в реальных системах, поиск кешированной трансляции в буфере TLB происходит при каждом обращении к памяти. Эмуляцию обращения к памяти можно разделить на две части: быстрый путь и медленный путь. Быстрый путь выполняется целиком в сгенерированном коде. В случае, если трансляция для адреса отсутствует в программном TLB или существуют какие-то особенности доступа к этой памяти, выполняется длинный путь, с передачей управления хелперу, который загрузит отсутствующую трансляцию в буфер TLB или обработает сложную ситуацию, и вернет управление обратно в транслированный код (или не вернет, в случае гостевого исключения). Для примера, рассмотрим транслированную в код хостовой системы x86-64 инструкцию загрузки переменной из памяти ldr архитектуры aarch64, показанную на рис 1.

В инструкциях 0-3 осуществляется поиск соответствующей адресу обращения записи в программном TLB. В младших битах адреса в записи хранятся флаги доступа, вроде TLB_DISCARD_WRITE (если страница доступна только на чтение), TLB_WATCHPOINT (если на странице установлена точка останова), TLB_MMIO (если на странице есть блок регистров устройств) и т. п. В инструкциях 4-5 адрес обращения подготавливается к сравнению с адресом, хранящимся в записи TLB. В 6-7 выполняется сравнение двух значений, и, если они отличаются, выполнение идет по медленному пути. Таким образом, в быстром пути выполняется доступ только к страницам, где ни один из флагов не установлен, то есть к нормальной памяти, без сторонних эффектов.

2.2.3 MMU контексты в Qemu

Для каждого виртуального процессора в Qemu существует сразу несколько виртуальных буферов TLB (MMU контекстов), трансляции в которых существуют одновременно друг с другом, и используются в разных режимах работы гостевого процессора. Разные контексты нужны для того, чтобы снизить количество очисток программного TLB из-за переключения режимов работы процессора. Как правило, гость использует несколько контекстов для своей работы. MMU контексты – это абстракция модели программного буфера TLB Qemu. Хотя и, в некоторой степени, похожи, контексты не эквивалентны режимам трансляции в aarch64. Также, контексты Qemu совсем не связаны с гостевым идентификатором процесса aarch64 (в разных MMU контекстах могут храниться трансляции как для разных процессов, так и для одного).

2.3 Описание оптимизации

Понимая, как работает программный MMU в Qemu, можно отметить основной его недостаток: поиск нужной трансляции в виртуальном TLB во время выполнения транслированного кода, работающего с памятью, вносит много накладных расходов (около 10 инструкций хостовой системы на одно обращение к памяти гостевой системы). А поскольку в RISC архитектурах, в среднем, больше трети от числа всех выполненных инструкций работают с памятью [3], замедление становится ощутимым. Предлагаемая оптимизация предполагает отказ от программного поиска нужной трансляции в виртуальном TLB, перекладывая работу по трансляции на аппаратный MMU хостовой системы. В

транслированном коде же остается, непосредственно, доступ к памяти с простой корректировкой адреса сложением, и, возможно, проверки над адресом, которые нельзя или нецелесообразно выполнять в таком сочетании гость/хост аппаратно. Достигается это путем размещения дополнительного вида на виртуальное адресное пространство гостевой системы, транслированного кода и инфраструктурного кода для его выполнения в одном адресном пространстве хостовой системы, как правило, в процессе эмулятора (рис. 2).

Рис. 1. Транслированный код для инструкции ldr.
Fig. 1. Translated code for ldr instruction.

По рис. 2 видно, что для использования подхода гостевое адресное пространство должно быть меньше, чем адресное пространство, доступное эмулятору. Размещение всего адресного пространства непрерывным куском требуется, чтобы избежать дополнительного программного уровня трансляции гостевого адреса при выделении его частями, что неприемлемо снизит потенциальный эффект ускорения. Размещение адресного пространства гостя в том же хостовом адресном пространстве, где работает транслированный код, необходимо из-за того, что переключение контекстов - относительно долгая операция, и положительный эффект от аппаратной трансляции адресов ее не перекроет.

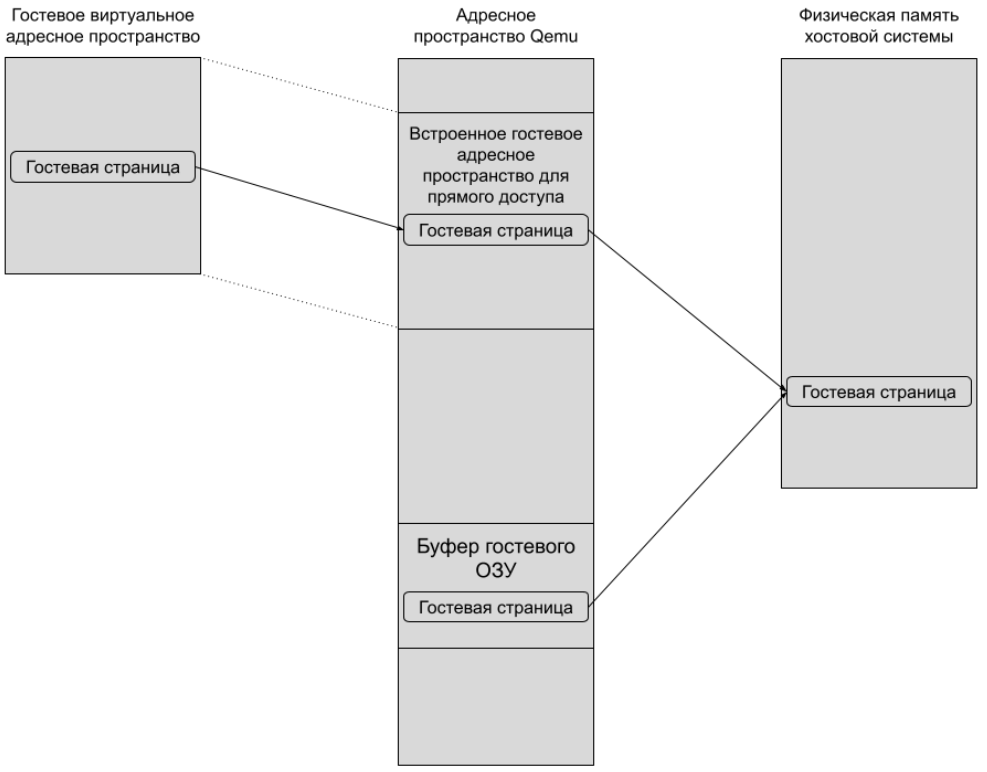

Рис. 2. Дополнительный вид на гостевое вирнуальное адресное пространство.
Fig. 2. Additional view of the guest virtual address space.

Общий алгоритм у всех подобных решений примерно следующий. Вначале, доступ ко всем страницам в диапазоне, выделенном для гостевых адресов, запрещен. Гостевая программ, обращаясь к какой-то странице, вызывает страничное исключение в хостовой системе, после чего эмулятор отображает страницу, вызвавшую исключение на страницу физической памяти хостовой системы, хранящую соответствующие данные физической памяти гостевой системы. Таким образом, следующий доступ из транслированного кода получит прямой доступ к необходимым данным, без необходимости искать трансляцию для адреса в программном TLB. Когда гостевая трансляция становится некорректной (из-за изменений в страничных таблицах), ее отображение на память хостовой системы удаляется. А значит, следующий доступ гостевого кода к этой странице снова вызовет исключение в хостовой системе и добавление уже актуального, корректного отображения.

Манипуляции с отображением страниц в системе – привилегированная операция и эмулятор, как пользовательский процесс, не имеет прямого доступа к таблицам трансляции. В работе [4] для модификации страничных таблиц хостовой системы использовали отдельный модуль ядра, который использовал внутренний интерфейс (Application Programming Interface, API) ядра для этой цели, и управлялся эмулятором. Основными недостатками подхода с модулем ядра является снижение отказоустойчивости системы и необходимость поддержки кода модуля, так как интерфейс ядра может меняться от версии к версии. В дальнейших работах, вместо модуля ядра, люди стали использовать, для той же цели, системный вызов `mmap` [3, 5]. В некоторых работах [6-7] транслированный код с отображенным адресным пространством гостевой системы запускают под аппаратной виртуализацией со вторым аппаратным уровнем трансляции адресов из гостевых физических в хостовые физические.

Такой подход обеспечивает больший контроль над выполнением кода и отображением адресов, но сильно увеличивает сложность разработки в виртуализированном окружении и подразумевает борьбу с замедлением, вызванным переключением процессора между режимами работы, поскольку большая часть функционала эмулятора остается работать без аппаратной виртуализации (из-за отсутствия инфраструктуры для этого под виртуализацией). Так же, для подобного исполнения в процессоре хостовой системы необходима поддержка аппаратной виртуализации со вторым уровнем трансляции [8].

В описанных исследованиях оптимизация используется для эмуляции 32-битных гостевых систем на x86-64 хостовой системе, у которой реальная разрядность виртуального адреса 48 бит. В работе [7] применили вариацию подхода для эмуляции системы архитектуры x86-64 на системе с архитектурой aarch64. Хотя в обеих архитектурах разрядность виртуального адреса одинаковая (48 бит), но в архитектуре aarch64 одновременно используются 2 таблицы страниц, то есть, реально, виртуальное адресное пространство вдвое больше, чем в x86-64, что оставляет достаточно места, чтобы структуры эмулятора не пересекались с адресами гостевой системы.

3. Особенности реализации

В нашем подходе мы решили использовать технику на основе страничных исключений и системного вызова `mmap`. К интересующей нас комбинации архитектур (aarch64 на x86-64) в общем виде оптимизация неприменима, так как процессу эмулятора доступна только младшая половина виртуальных адресов хостовой системы, то есть 47 бит. Часть адресов использует сам эмулятор и библиотеки, при компактном расположении которых, для виртуального адреса гостевой системы остается 46 бит. В гостевой системе с архитектурой aarch64 напротив корректных адресов в 2 раза больше, чем 48 бит (за счет использования двух страничных таблиц).

Однако, часто гостевые ОС (например, на базе Linux) делят адресное пространство aarch64 на старшую половину адресов для привилегированного режима и младшую для пользовательского и не используют их одновременно. К тому же, архитектура aarch64 позволяет отключать часть уровней таблиц трансляции, что эффективно уменьшает разрядность адреса. Например, отключение одного уровня таблиц оставляет разрядность в 39 бит, что позволяет адресовать 512 ГиБ памяти, достаточных для большинства задач. К тому же, это стандартная конфигурация ядра Linux при четырехкилобайтных страницах.

Поэтому принято допущение, что гостевое ПО одновременно использует только одну половину виртуального адресного пространства (старшую или младшую) разрядностью 39 бит, что позволяет выделить немного меньше 2^8 диапазонов под гостевые виртуальные адресные пространства в адресном пространстве эмулятора, оставляя запас для дополнительных оптимизаций.

Архитектура решения допускает, что гостевое ПО может использовать любой адрес в своей работе, не только из ожидаемого диапазона, но в такой ситуации обращение будет обработано через медленный путь, хотя, потенциально, могло быть осуществлено быстрее через быстрый путь при использовании программного буфера TLB. При большом количестве таких адресов, скорость эмуляции может упасть даже больше, чем при отключенной оптимизации, так что стоит избегать подобных ситуаций.

3.1 Аппаратный буфер TLB

Для отображения гостевых виртуальных страниц на хостовые в Qemu, в дополнение к программному TLB добавлен аппаратный TLB. Термин “аппаратный TLB” используется для контраста с программным TLB, его не следует путать с реальным буфером TLB хостовой

системы, который тоже участвует в работе аппаратного буфера TLB эмулятора Qemu. Аппаратный TLB работает параллельно программному TLB и хранит трансляции для страниц с нормальной памятью без сторонних эффектов, к которым возможен непосредственный доступ из транслированного кода. Отключить программный TLB при этом нельзя, так как, помимо транслированного кода, трансляция адресов также требуется кодогенератору, периферийным устройствам, отладчику и пр., и отсутствие кеша трансляций в этих случаях значительно замедлит эмуляцию.

Аппаратный TLB, как и программный, обеспечивает добавление записей в буфер TLB, удаление сразу всех записей или постранично. Для прямого отображения гостевых виртуальных страниц на буфер с данными гостевой памяти используется системный вызов `mmap` с соответствующими флагами доступа (PROT_READ, PROT_WRITE). Для удаления трансляции из аппаратного TLB используется `mmap` с флагом PROT_NONE (то есть обращение к странице памяти вызовет исключение).

Интересной особенностью такого распределенного между таблицами трансляции и реальным TLB системы виртуального TLB является его, условно бесконечный размер (отсутствует вытеснение старых трансляций из-за добавления новых трансляций с таким же ключом). Это позволяет, иногда, избегать повторных избыточных трансляций, в отличие от программного TLB.

3.2 Модификация кодогенерации

Быстрый путь, при программном TLB выполняет несколько операций:

1) проверка, что трансляция существует, и доступ разрешен;
2) проверяет, что доступ выровненный;
3) корректирует виртуальный гостевой адрес для получения виртуального хостового (используя информацию в программном TLB);
4) осуществляет доступ к памяти.

Для аппаратного TLB шаги другие:

1) проверить, что адрес не выходит за границы поддерживаемого диапазона;
2) проверить, что доступ выровнен;
3) прибавить адрес к базе диапазона в адресном пространстве эмулятора;
4) выполнить доступ к памяти.

Первый пункт необходим, потому что гостевое ПО может использовать любое 64-битное значение в качестве адреса, и обратиться к запрещенной для него памяти. Проверку на выравнивание доступа можно совместить, в случае младших адресов, с проверкой адреса. Для старших гостевых адресов – это отдельные инструкции. Адрес базы диапазона прибавляется в x86-64 одновременно с доступом к памяти, однако, адрес базы необходимо перед этим загрузить в регистр. Во время доступа к памяти аппаратно проверяется, что трансляция для страницы существует, и доступ к ней разрешен. Во всех пунктах, если условие не удовлетворяется, происходит переход к медленному пути (через ветвление или исключение).

На рис. 3 показан транслированный код для инструкции `ldr` со включенным аппаратным TLB.

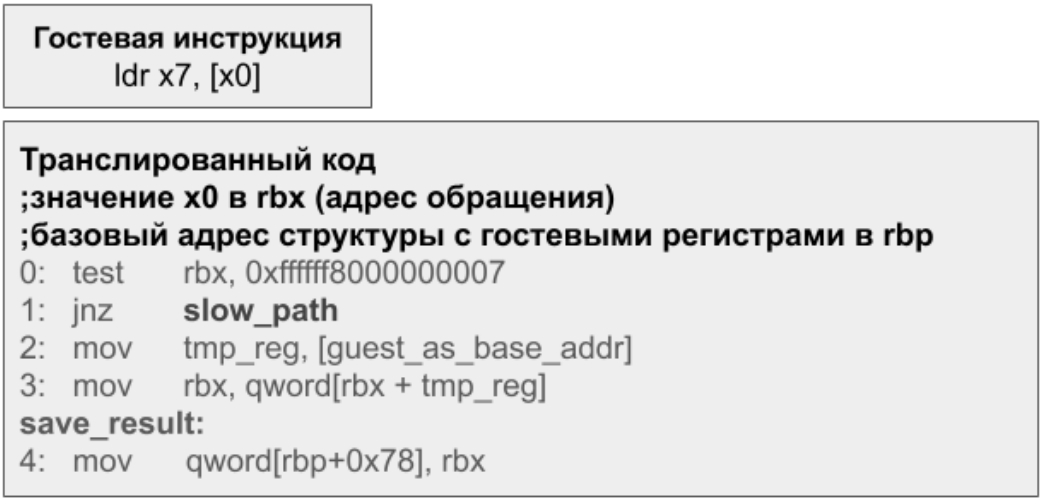

Рис. 3. Транслированная инструкция `ldr` *для использования с аппаратным TLB.*
Fig. 3. Translated `ldr` *instruction for hardware TLB usage.*

3.3 Отключение проверок в транслированном коде

Хотя наличие в быстром пути проверок на выравнивание адреса и попадание адреса в ожидаемый диапазон обеспечивает корректность эмуляции и стабильность работы эмулятора, замечено, что на правильно работающих программах эти проверки никогда не срабатывают. Поэтому, если потребность в ускорении перекрывает потенциальную, но маловероятную, некорректность и нестабильность работы эмулятора, проверки можно исключить из быстрого пути, сократив его. Результат удаления проверок из быстрого пути для инструкции `ldr` показан на рис. 4.

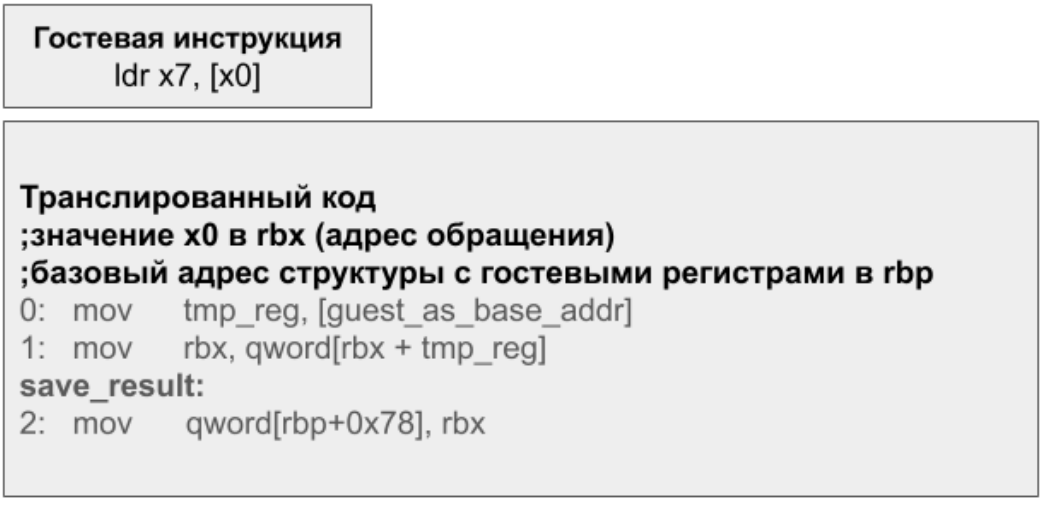

Рис. 4. Код инструкции `ldr` *для аппаратного TLB без проверок.*
Fig. 4. Hardware TLB `ldr` *code without checks.*

3.4 Использование индекса MMU для избирательного применения оптимизации

Поскольку медленный путь, при использовании подхода становится еще медленнее, чем с программным MMU (за счет обработки через прерывание, вместо простого вызова процедуры), сценарии с большим количеством медленных путей плохо подходят для алгоритма. Часто, работа через медленный путь вызвана обращением гостевых программ к устройствам периферийным устройствам. В [3] было замечено, что программы не имеют прямого доступа к устройствам, а делают это посредством привилегированного ПО, которое работает в режиме ядра. Значит, можно не включать оптимизацию для кода ядра гостевой системы, а продолжать использовать модуль программного MMU эмулятора, а для пользовательского гостевого кода использовать аппаратный буфер TLB. К тому же, код ядра обычно располагается в старшей части адресного пространства, а, как упомянуто ранее, транслированный код для доступа к старшим адресам длиннее, чем к младшим, из-за более длинной проверки и корректировки гостевого виртуального адреса, поэтому потенциальное ускорение выше при обращении к младшим адресам, а для старших адресов сходит на нет.

Поскольку решить, использовать оптимизацию или нет надо на этапе трансляции кода, в качестве критерия используется MMU индекс, который с достаточной точностью позволяет отличать ядерный и пользовательский режим работы.

Другим примером исключения MMU индекса из работы алгоритма могут служить индексы, которые не используются из сгенерированного кода, а только хелперами и инфраструктурой эмулятора. Нет смысла поддерживать аппаратный TLB для такого индекса, поскольку он никогда не будет использоваться.

3.5 Отслеживание гостевых контекстов

Внесение изменений в аппаратный TLB - относительно дорогая операция, в сравнении с модификацией программного TLB. Поэтому, если выполнять очистку аппаратного TLB каждый раз, когда гостевая система переключает процесс, как это делает программный TLB в Qemu, накладные расходы на очистку и повторную трансляцию адресов будут неоправданными. Для сглаживания отрицательного эффекта от переключения процессов в аппаратный TLB добавлено отслеживание идентификатора процесса, по типу того, как это происходит в реальном TLB современных машин. При смене таблиц трансляций во время переключения контекстов, трансляции предыдущего процесса не отбрасываются, а сохраняются для будущего использования в пуле контекстов до момента, когда процесс станет активен снова.

Для отслеживания текущего процесса используется, как и в реальном TLB, архитектурно определенный в системе команд aarch64 идентификатор процесса `asid`. Кешированные трансляции процесса удаляются при очистках кеша, не связанных с переключением контекста. В случае, отсутствия свободных контекстов в пуле контекстов, самый старый контекст заменяется новым.

Пул контекстов формируется из доступных диапазонов гостевых виртуальных адресных пространств. Для 39-битного гостя и эмулятора на хостовой системе x86-64, максимальный размер пула для каждого виртуального процесса, при многопоточной работе кодогенератора: $\texttt{floor}(2^8/N_{cpus})$, где `floor` – функция округления вниз, а N_{cpus} – количество виртуальных процессоров. В случае с однопоточной кодогенерацией, максимальный размер пула немногим меньше 2^8. Однако при работе реальных гостевых систем он не используется целиком, потому что полная очистка буфера (`tlb flush`) случается намного чаще, чем переключение между 2^8 уникальными гостевыми контекстами, а в этом случае все гостевые адресные диапазоны возвращаются обратно в пул. По результатам наблюдений, оптимальный размер пула на поток кодогенератора находится в диапазоне от 8 до 16.

4. Производительность

Для оценки производительности предложенного решения в качестве гостевой системы была выбрана виртуальная машина raspberry 3b [9] (одноплатный компьютер с 4 ядерным SoC с архитектурой aarch64, 1 Гб ОЗУ и набором периферии). В качестве гостевой ОС использовалась Raspberry Pi OS Lite [10] (основана на ОС Debian 11 Bullseye). Параметры хостовой системы: Intel® Core™ i7-8700 CPU @ 3.20GHz, 32Гб ОЗУ, 500Гб SSD.

Для тестовых примеров использовались несколько реальных сценариев работы: загрузка ОС, скачивание файла из сети, установка программы, архивация/деархивация файла, а также эталонный тест Nbench. Для большей достоверности, результаты измерений подвергались статистической обработке. Nbench измеряет производительность в итерациях в секунду. В остальных же тестах измерялось время работы, но для унифицированного представления полученные значения инвертировались. Результаты измерений представлены в табл. 1.

Табл. 1. Сравнение производительности базовой версии Qemu и с включенной оптимизацией.
Table 1. Performance comparison of vanilla Qemu and Qemu with abovementioned optimization.

Название теста	Базовая Qemu 9.1.50	Qemu 9.1.50 с аппаратным TLB	Производительность (больше 100 – ускорение, меньше – замедление), %
Загрузка ОС	22.02	18.69	84.87
Скачивание файла ~50 МБ	3.80	3.81	100.26
Установка программы	11.01	10.83	98.36
Многопоточное сжатие файла ~50 МБ	15.67	34.12	217.74
Извлечение файла ~50 МБ	28.24	30.76	108.92
Make (программы nbench)	17.36	16.31	93.95
Nbench общее	1.91	3.11	162.82
NUMERIC SORT (Nbench)	185.79	504.01	271.27
STRING SORT (Nbench)	65.18	107.13	164.36
FP EMULATION (Nbench)	128.22	284.59	221.95
FOURIER (Nbench)	6009.23	6608.07	109.96
ASSIGNMENT (Nbench)	9.03	11.51	127.46
IDEA (Nbench)	3122.83	3788.79	121.32
HUFFMAN (Nbench)	1082.52	1355.60	125.22
NEURAL NET (Nbench)	4.68	5.36	114.52
LU DECOMPOSITION (Nbench)	190.67	220.29	115.53
INTEGER INDEX (Nbench)	27.94	44.59	159.59
FLOATING-POINT INDEX (Nbench)	7.89	9.04	114.57
MEMORY INDEX (Nbench)	6.48	9.15	141.20
BITFIELD (Nbench)	204681940	253949410	124.07

По измерениям видно, что производительность со включенной оптимизацией зависит от типа нагрузки на гостевую систему. Большое ускорение получено при работе бенчмарков (ускорение до 2.71 раз, 1.62 раза в среднем), и на мультипоточном сжатии файла (2.17 раз). В остальных реальных примерах производительность изменилась незначительно, кроме теста с

загрузкой ОС, где производительность снизилась на 15%. Замедление при загрузке можно объяснить тем, что эффект ускорения от использования аппаратного TLB не перевесил добавившиеся накладные расходы на его работу.

5. Заключение

В статье рассмотрена оптимизация трансляции виртуальных адресов при кроссплатформенной полносистемной эмуляции, позволяющая перенести часть нагрузки с программного MMU эмулятора на аппаратуру трансляции адресов хостовой системы. Выполнена практическая реализация описанного подхода на эмуляторе Qemu. Замеры производительности итогового решения показывают большое ускорение при использовании оптимизации, в том числе и для реальных программ, а не только бенчмарков.

Предложенное исполнение аппаратного TLB для ускорения трансляции гостевых адресов может применяться для эмуляции aarch64 на x86-64, что раньше в работах не встречалось.

Хотя описанный подход нельзя назвать универсальным, потому что он предполагает определенные настройки гостевой операционной системы, часто такая настройка возможна, либо уже используется в выбранной системе.

Для снятия ограничения на 3 уровня таблиц трансляций в гостевой системе возможно использование расширение процессоров x86-64, увеличивающего разрядность виртуального адрес хостовой системы до 57 бит [11]. Это обеспечит достаточный запас виртуальных адресов для применения описанного выше подхода для систем aarch64 с полным 48-битным адресом. В перспективе мы собираемся проверить этот подход на практике.

Список литературы / References

[1]. Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for Cross-ISA system mode emulation. In Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '14). Association for Computing Machinery, New York, NY, USA, 117–128.

[2]. “Qemu – a generic and open source machine emulator and virtualizer,” https://www.qemu.org/, accessed: 22.05.2025.

[3]. Antoine Faravelon. Acceleration of memory accesses in dynamic binary translation. Operating Systems [cs.OS]. Université Grenoble Alpes, 2018. English.

[4]. Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In ACM SIGPLAN Notices, volume 49, pages 117–128. ACM, 2014.

[5]. Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, WeiChung Hsu, Bin Li, and Yong Guan. Hspt: Practical implementation and efficient management of embedded shadow page tables for cross-isa system virtual machines. In ACM SIGPLAN Notices, volume 50, pages 53–64. ACM, 2015.

[6]. Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated crossarchitecture full-system virtualization. ACM Transactions on Architecture and Code Optimization (TACO), 13(4):36, 2016.

[7]. S. Rodzevich, K. Batuzov, D. Koltunov, A. Cheremnov and I. Shlyapin, "Efficient MMU Emulation in Case of Cross-ISA Dynamic Binary Translation," 2024 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russian Federation, 2024, pp. 1-6, doi: 10.1109/ISPRAS64596.2024.10899135.

[8]. “AMD-V Nested Paging White Paper,” https://www.cse.iitd.ac.in/~sbansal/csl862-virt/readings/NPT-WP-1%201-final-TM.pdf, accessed: 22.05.2025.

[9]. “Raspberry Pi 3 Model B,” https://www.raspberrypi.com/products/raspberry-pi-3-model-b/, accessed: 22.05.2025.

[10]. “Raspberry Pi OS (Legacy) Lite,” https://downloads.raspberrypi.com/raspios_oldstable_lite_arm64/images/raspios_oldstable_lite_arm64-2025-05-07/2025-05-06-raspios-bullseye-arm64-lite.img.xz, accessed: 22.05.2025.

[11]. “Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux,” https://lenovopress.lenovo.com/lp1468-introduction-to-5-level-paging, accessed: 22.05.2025.

Информация об авторах / Information about authors

Дмитрий Николаевич ПОЛЕТАЕВ – разработчик программного обеспечения, сотрудник Института системного программирования РАН. Сфера научных интересов: обратная разработка, анализ бинарного кода, виртуальные машины.

Dmitry Nikolaevich POLETAEV – software developer at Ivannikov Institute for System Programming of the Russian Academy of Sciences. Research interests: reverse engineering, binary code analysis, virtual machines.

Павел Михайлович ДОВГАЛЮК – инженер, кандидат технических наук. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, отладчики, эмуляторы.

Pavel Mikhailovich DOVGALYUK – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

Георгий Николаевич ТЕЙС – инженер по тестированию Института системного программирования РАН с 2022 года. Сфера научных интересов: автоматизация процессов в среде системного программирования.

Georgiy Nikolaevich TEYS – QA engineer of the Institute for System Programming of the RAS since 2022. His research interests include processes automation in system programming.

Максим Алексеевич КОСТИН – инженер отдела компиляторных технологий ИСП РАН. Научные интересы включают эмуляторы, динамическую двоичную трансляцию, оптимизации.

Maksim Alekseevich KOSTIN – engineer at Compiler Technology department of ISP RAS. His research interests include emulators, dynamic binary translation, optimizations.