DOI: 10.15514/ISPRAS-2026-38(2)-9

Оптимизация передачи управления в полносистемных эмуляторах

[1] *М.А. Костин, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>*
[1,2] *П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *Д.Н. Полетаев, ORCID: 0009-0005-8872-2802 <poletaev@ispras.ru>*
[1] *Г.Н. Тейс, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>*
[1] *Н.И. Фурсова, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Новгородский государственный университет им. Ярослава Мудрого, Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.*

Аннотация. Полносистемные эмуляторы позволяют имитировать работу всей машины, включая процессор и периферию. Данная технология значительно упрощает разработку и отладку программ под различные платформы, так как она избавляет от необходимости иметь в распоряжении соответствующую аппаратуру. Полносистемные эмуляторы в основном используют динамическую двоичную трансляцию для приемлемой производительности. Целью данной работы является повышение производительности динамической двоичной трансляции путём применения оптимизаций передачи управления. В статье проводится обзор различных оптимизаций с учётом ограничений полносистемных эмуляторов, в том числе рассматривается проблема межстраничных переходов. Для реализации были выбраны межстраничное связывание и программное предсказание. При анализе существующих решений доказывается, что один из методов проверки межстраничных переходов неверен, а для корректного метода предлагается улучшенный алгоритм, составленный с учётом особенностей гостевой архитектуры. Помимо этого, в статье приводится новый подход к реализации программного предсказания в полносистемных эмуляторах, который упрощает проверочный код и увеличивает точность предсказания. Данные оптимизации были реализованы в полносистемном эмуляторе QEMU. Тесты на двух разных машинах показали, что производительность QEMU была улучшена на 23,5% и 21,1% в среднем, а максимум улучшения составил 89,9% и 76,9% соответственно.

Ключевые слова: полносистемная эмуляция; динамическая двоичная трансляция; оптимизация программ; передача управления; QEMU.

Для цитирования: Костин М.А., Довгалюк П.М., Полетаев Д.Н., Тейс Г.Н., Фурсова Н.И. Оптимизация передачи управления в полносистемных эмуляторах. Труды ИСП РАН, том 38, вып. 2, 2026 г., стр. 129–148. DOI: 10.15514/ISPRAS–2026–38(2)–9.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022.

Optimizing control transfer in full-system emulators

[1] *M.A. Kostin, ORCID: 0009-0002-1464-8302 <maksim.kostin@ispras.ru>*
[1,2] *P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *D.N. Poletaev, ORCID: 0009-0005-8872-2802 <poletaev@ispras.ru>*
[1] *G.N. Teys, ORCID: 0009-0008-6059-2315 <george.teys@ispras.ru>*
[1] *N.I. Fursova, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Yaroslav-the-Wise Novgorod State University, 41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.*

Abstract. Full-system emulators replicate an entire machine's operation, including its CPU and peripherals. This technology significantly simplifies software development and debugging for various platforms, as it eliminates the need for the corresponding physical hardware. Full-system emulators usually use dynamic binary translation for acceptable performance. The aim of this work is to improve the performance of dynamic binary translation by applying control transfer optimizations. The paper provides an overview of various optimizations, taking into account the constraints of full-system emulators, including cross-page problem. Cross-page block linking and software prediction were chosen for implementation. When analyzing existing solutions, it is proven that one of the methods for cross-page branch validation is incorrect. An improved algorithm, developed with consideration for the specifics of the guest architecture, is proposed for the correct one. Furthermore, the paper proposes a novel approach for software prediction implementation for full-system emulators that keeps the check code simple and increases prediction accuracy. These optimizations were implemented in QEMU full-system emulator. Benchmarks on two different machines showed that QEMU's performance improved by 23.5% and 21.1% on average, with peak improvements reaching 89.9% and 76.9%, respectively.

Keywords: full-system emulation; dynamic binary translation; program optimization; control transfer; QEMU.

For citation: Kostin M.A., Dovgalyuk P.M., Poletaev D.N., Teys G.N., Fursova N.I. Optimizing control transfer in full-system emulators. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 2, 2025, pp. 129-148 (in Russian). DOI: 10.15514/ISPRAS-2026-38(2)-9.

Acknowledgements. This work is supported by Russian Science Foundation under grant no. 24-11-20022.

1. Введение

Полносистемные эмуляторы позволяют имитировать работу всей машины, включая процессор и периферию, при этом процессор может быть другой архитектуры. Данная технология значительно упрощает разработку и отладку программ под различные платформы, так как она избавляет от необходимости иметь в распоряжении соответствующую аппаратуру. Помимо этого, полносистемные эмуляторы предоставляют большое количество информации о состоянии машины во время исполнения, позволяют его сохранять и воссоздавать, что делает их основополагающим инструментом для анализа программного обеспечения. Обычно для исполнения кода эмулируемой машины они используют динамическую двоичную трансляцию.

Двоичная трансляция – это технология, заключающаяся в переводе последовательности машинных инструкций *A* в последовательность машинных инструкций *B* с определённым набором требований, например:

- *B* должна делать то же, что и *A*, но исполняться на процессоре с другой архитектурой;

- *B* должна делать то же, что и *A*, но включать в себя вставки анализирующих процедур, отслеживающих событие *C*.

Последовательность машинных инструкций, то есть код, который подвергается двоичной трансляции, называется гостевым. А код, получаемый в результате трансляции – целевым.

Архитектуры, чьим наборам команд принадлежат инструкции гостевого и целевого кода, называются гостевой и целевой соответственно.

Если двоичная трансляция проводится во время исполнения, то её называют динамической. Полносистемные эмуляторы используют динамическую двоичную трансляцию для имитации работы целой операционной системы, включая библиотеки и пользовательские приложения. В таком случае очень важна производительность.

Обычно в условиях динамической двоичной трансляции исполняемый код делится на блоки, которые транслируются и исполняются по отдельности. Будем называть их блоками трансляции. Оттранслированные блоки заносятся в кэш, чтобы в дальнейшем их можно было заново использовать без повторной трансляции.

Часто блоки трансляции завершаются инструкцией передачи управления. После исполнения одного из таких блоков, транслятору нужно найти следующий, основываясь на текущем состоянии виртуального процессора. Базовое решение заключается в выходе из оттранслированного кода в главный цикл, где будет найден следующий блок, но это очень дорогостоящая операция. Поэтому для повышения производительности используются различные оптимизации, но в полносистемных эмуляторах их применение усложняется определёнными ограничениями.

Целью данной работы является повышение производительности динамической двоичной трансляции путём применения оптимизаций передачи управления.

В разделе 2 проводится обзор различных оптимизаций передачи управления с учётом ограничений полносистемных эмуляторов, рассматривается проблема межстраничных переходов и анализируются существующие решения. В разделе 3 описывается реализация выбранных оптимизаций в QEMU [1], а также приводится ряд улучшений, упрощающих генерацию кода. Наконец, в разделе 4 приводятся результаты тестов производительности и их анализ.

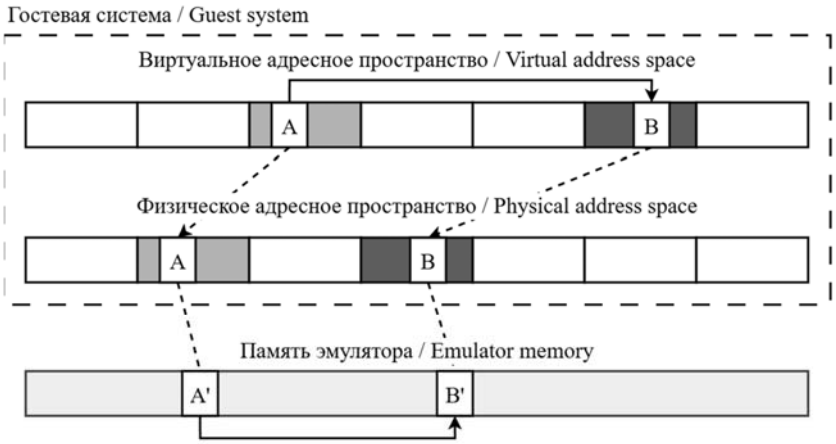
2. Обзор оптимизаций передачи управления

Для повышения производительности передачи управления используются различные оптимизации в зависимости от типа инструкции.

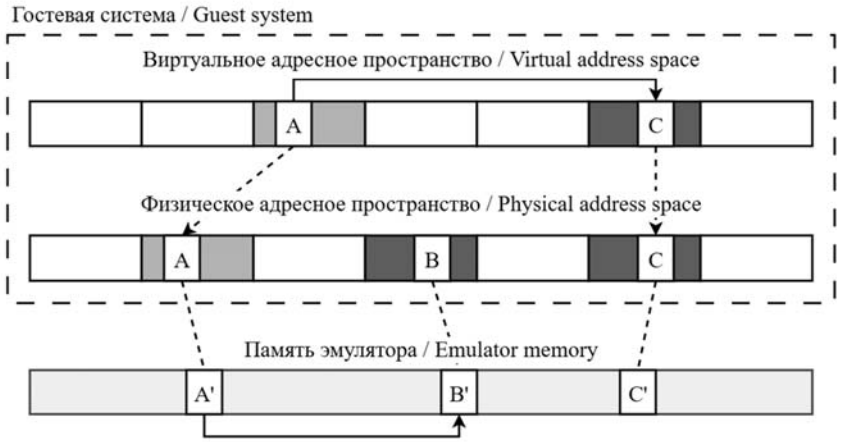
2.1 Прямые переходы

Передача управления по постоянному, заранее известному адресу называется прямым переходом. Основные издержки обработки прямых переходов вызваны поиском блока трансляции, соответствующего счётчику инструкций и другому состоянию виртуального процессора. Эти параметры однозначно определяют последующий блок трансляции, но проблема заключается в том, что сам блок может быть ещё не оттранслированным.

Для того, чтобы каждый раз, встречая инструкцию прямого перехода, не совершать дорогостоящую операцию выхода в главный цикл для поиска или трансляции блока, применяется *связывание блоков* [1, 2]. Гостевая инструкция прямого перехода транслируется в аналогичную инструкцию из целевого набора команд, пока что ничего не делающую. Далее, единожды осуществив выход в главный цикл и получив необходимый блок трансляции, производится запись соответствующего адреса в ранее оттранслированную инструкцию. Проверку состояния процессора добавлять не нужно, так как оно уже было учтено при поиске блока трансляции.



(б) Прямые переходы в оттранслированном и гостевом коде эквивалентны
(b) Direct branches in translated and guest code are equivalent



(б) Прямые переходы в оттранслированном и гостевом коде отличаются
(b) Direct branches in translated and guest code differ

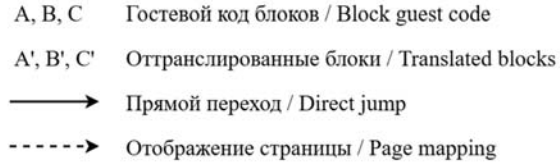


Рис. 1. Проблема межстраничных переходов.
Fig. 1. Cross-page branch problem.

2.1.1 Проблема межстраничных переходов

Так как полносистемные эмуляторы сами выполняют трансляцию адресов, то они в основном используют физическую индексацию блоков [1, 2], то есть блоки индексируются своими физическими адресами, а не виртуальными. Данная особенность накладывает определённое ограничение на связывание блоков. Рассмотрим рис. 1(а), оттранслированные блоки A’ и B’ связаны, и прямой переход в оттранслированном коде соответствует прямому переходу в гостевом коде. Затем, отображение виртуальной страницы с гостевым кодом B изменяется, как показано на рис. 1(б). Так как сменилось лишь отображение страницы, то физическая страница с гостевым кодом B не была модифицирована, а блоки трансляции индексируются именно физическими адресами, таким образом блок B’ остался валидным. В следствие, связь между оттранслированными блоками A’ и B’ сохранилась. Тогда последующий прямой переход от A к C будет выполнен эмулятором неправильно, поскольку управление будет передано блоку трансляции B’ вместо C’. Аналогичная ситуация может произойти, если отображение страницы с гостевым кодом B будет удалено. При переходе, вместо генерации исключения, будет выполнен блок трансляции B’. Получается, что межстраничное связывание блоков в обоих случаях приводит к некорректному выполнению гостевого кода.

Для преодоления данного ограничения межстраничные переходы должны обрабатываться специальным образом [2, 3], что будет подробно рассмотрено в подразделе 2.3.

2.2 Косвенные переходы

Передача управления по гостевому адресу, неизвестному на момент трансляции, например, по значению регистра, называется косвенным переходом. Основные издержки обработки таких переходов, также как и прямых, вызваны поиском блока трансляции, но в данном случае задача усложняется тем, что цель перехода может быть далеко не одна.

Существует множество оптимизаций обработки косвенных переходов [4], некоторые из которых описаны далее, а их псевдокод приведён в табл. 1.

Табл. 1. Псевдокод различных оптимизаций обработки косвенных переходов.
Table 1. Pseudocode of various indirect branch handling optimizations.

Оптимизация	Хэш-таблица	Стек адресов возврата	Программное предсказание

Псевдокод	*State* := State(*CPU*)
	Hash := Hash(*State*)
	Entry := *Table*[*Hash*]
	if *State* = *Entry.State*:
	  goto *Entry.HPC*
	goto *SlowPath*
		Caller:
		  ...
		  push *GRA*, *HRA*
		  goto *Callee*
		HRA:
		  goto *Target*
		
		Callee:
		  ...
		  pop *GRA*, *HRA*
		  if *GPC* = *GRA*:
		  goto *HRA*
		  goto *SlowPath*
			if *GPC* = GPC_0:
			  goto HPC_0
			if *GPC* = GPC_1:
			  goto HPC_1
			...
			if *GPC* = GPC_N:
			  goto HPC_N
			goto *SlowPath*

Обычно трансляторами используются *хэш-таблицы*, чтобы ускорить поиск следующего блока. Часто код поиска вписывают прямо в блок вместо вызова обработчика. Однако, если проверка подразумевает сравнение не только гостевого счётчика инструкций (*GPC*), но и остального состояния виртуального процессора, которое ещё нужно заранее получить, код

поиска значительно усложняется. Данная проблема в первую очередь касается полносистемной эмуляции.

Инструкции возврата из процедуры могут быть оптимизированы введением *теневого стека вызовов*. При вызове функции гостевой и целевой адреса возврата (*GRA* и *HRA*, соответственно) заносятся в стек, а при выходе из неё извлекаются. Если гостевой счётчик инструкций совпадает с гостевым адресом возврата, то управление передаётся по целевому адресу возврата. Количество инструкций целевого кода сокращается относительно поиска по хэш-таблице, да и точность выше, но данная оптимизация подразумевает большое число операций с памятью, а также требует грамотной обработки ситуаций с переполнением и опустошением стека.

Программное предсказание заключается во вставке перед вызовом обработчика цепочки предсказаний, каждое из которых состоит из двух частей: сравнение и условный переход. Этап сравнения представляет собой проверку гостевого счётчика инструкций (*GPC*) на равенство гостевому адресу цели (GPC_i). В случае успеха, управление передаётся соответствующему блоку трансляции (по целевому адресу HPC_i), иначе проводится следующее предсказание.

Преимущество программного предсказания над остальными оптимизациями в том, что переходы являются прямыми, в следствие чего отсутствует необходимость в проверках состояния виртуального процессора. Также, в отличие от теневого стека вызовов, программное предсказание не требует интенсивной работы с памятью и может быть применено ко всем косвенным переходам, а не только инструкциям возврата из процедуры. Но, чтобы преимущество сохранилось и в условиях полносистемной эмуляции, необходимо эффективно обрабатывать межстраничные переходы.

2.3 Применение в полносистемных эмуляторах

В Embra [2] для прямых переходов применяется связывание блоков, а проблема межстраничных переходов решена следующим образом: перед каждым блоком вставляется проверочный код, который сравнивает физическую страницу, соответствующую текущему счётчику инструкций, с физической страницей, соответствующей цели перехода. В случае совпадения исполнение блока продолжается, иначе управление передаётся главному циклу транслятора. Для косвенных переходов в Embra применяется *спекулятивное связывание*, когда блок связывается с первой встреченной целью. Для предотвращения некорректного выполнения кода, перед каждым блоком вставляется ещё один проверочный код, сравнивающий текущий счётчик инструкций с виртуальным адресом блока. Чтобы снизить накладные расходы, для каждого типа связывания используется своя точка входа. Таким образом, проверочный код выполняется только тогда, когда это необходимо. Данная оптимизация фактически является программным предсказанием с длиной цепочки равной единице.

В работе [3] был представлен более простой метод проверки, заключающийся в сравнении виртуальных страниц, а не физических. В ней предлагается отслеживать виртуальное адресное пространство при помощи виртуального *iTLB* (*instruction TLB*), каждая запись которого включает в себя лишь номер виртуальной страницы. Тогда при межстраничном переходе можно сравнивать соответствующую текущему счётчику инструкций запись из *iTLB*, с виртуальной страницей, соответствующей цели перехода. При этом, проверка является "ленивой", то есть проводится непосредственно перед переходом.

На самом деле, данное решение использовать нельзя, так как оно не способно правильно обработать ситуацию, когда отображение виртуальной страницы сменяется на другую физическую страницу. Чтобы доказать данное утверждение, авторы провели эксперимент в эмуляторе QEMU [1]. Для этого в эмуляторе был реализован виртуальный *iTLB*, и подготовлена гостевая машина с архитектурой AArch64. Машина находилась под

управлением ядра Linux. Для демонстрации была написана тестовая программа, её исходный код на языке программирования C приведён в листинге 1.

```
#include <assert.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    size_t page_size = 4096;
    int prot = PROT_READ | PROT_WRITE | PROT_EXEC;
    int flags0 = MAP_PRIVATE;
    int flags1 = MAP_PRIVATE | MAP_FIXED;
    int source_fd = open("source.bin", O_RDONLY, S_IRWXU);
    int target0_fd = open("target0.bin", O_RDONLY, S_IRWXU);
    int target1_fd = open("target1.bin", O_RDONLY, S_IRWXU);
    void *source = mmap(NULL, page_size * 2, prot, flags0, source_fd, 0);
    void *target = mmap(source + page_size, page_size, prot, flags1,
                        target0_fd, 0);
    int (*execute)(void) = source;

    execute();
    execute();
    mmap(target, page_size, prot, flags1, target1_fd, 0);
    execute();
    assert(execute() == 1);
}
```

Листинг 1. Исходный код тестовой программы на языке программирования C.
Listing 1. Source code of the test program in the C programming language.

```
source.bin:
   0:   14000400   b   0x1000

target0.bin:
   0:   52800000   mov w0, #0x0
   4:   d65f03c0   ret

target1.bin:
   0:   52800020   mov w0, #0x1
   4:   d65f03c0   ret
```

Листинг 2. Вывод дизассемблера для отображаемых в память файлов.
Listing 2. Disassembler output for memory-mapped files.

Программа отображает два файла на разные виртуальные страницы в адресном пространстве процесса. Файлы содержат машинный код, приведённый в листинге 2. Отображение осуществляется таким образом, что код *source* содержит в себе прямой межстраничный переход к *target*. Далее, программа дважды осуществляет вызов по адресу

source, чтобы эмулятор связал блоки трансляции, соответствующие *source* и *target0*. Затем, по адресу *target* вместо *target0* отображается *target1*. Ещё один дополнительный вызов осуществляется для обновления *iTLB*, так как запись о странице была сброшена при смене отображения. Поскольку *target1* возвращает 1, то последний вызов по адресу *source* должен вернуть 1. То есть, если ситуация с изменением отображения страницы будет обработана неправильно, вместо *target1* управление будет передано *target0*, и вызов вернёт 0, *assert* провалится, и программа аварийно завершит выполнение.

Выполнение программы в оригинальном QEMU завершилось успешно, а в QEMU с виртуальным iTLB – аварийно. Таким образом, данный метод не предотвращает некорректное выполнение кода, и его нельзя использовать. Связано это с тем, что недостаточно просто удостовериться в наличии отображения виртуальной страницы, необходимо также учитывать соответствующую ей физическую страницу, как изначально было предложено в работе [2]. То есть, в *iTLB* необходимо также записывать номера физических страниц, и при межстраничном прямом переходе проверять их на совпадение с физической страницей цели.

3. Реализация

Реализация оптимизаций будет проводиться в полносистемном эмуляторе с открытым исходным кодом QEMU [1] версии 9.1 под конкретную пару архитектур. Гостевой архитектурой является AArch64, а целевой – x86-64. И гостевая, и целевая машины находятся под управлением ядра Linux. Ядро гостевой настроено на трёхуровневые таблицы трансляции с размером страницы 4 килобайта, таким образом, размер виртуального адреса составляет 39 бит и для пользовательского пространства, и для пространства ядра [5]. Размер физического адреса гостевой машины не превышает 48 бит [6].

Для имитации работы операционной системы и пользовательских приложений QEMU использует динамическую двоичную трансляцию. Её единицей является блок трансляции с одним входом и несколькими выходами. Каждому блоку трансляции соответствует некоторая информация: адрес физической страницы, цели переходов, состояние виртуального процессора и т.д. При этом для многих гостевых архитектур, включая AArch64, блоки не привязаны к конкретному виртуальному адресу, что позволяет заново их использовать в различных виртуальных адресных пространствах.

В QEMU уже реализовано связывание блоков, но оно не применяется для межстраничных переходов. Для межстраничных прямых и всех косвенных переходов вызывается обработчик поиска блока через хэш-таблицу. Это помогает избежать выхода из оттранслированного кода, но всё ещё занимает значительное время. Например, при загрузке гостевой операционной системы обработчик поиска блока без учёта затрат на вызов занимает примерно 21% времени выполнения оттранслированного кода.

3.1 Межстраничное связывание блоков

Для проведения проверки необходимо обладать следующей информацией: номер текущей виртуальной страницы и номер физической страницы, на которую она была отображена в момент связывания блоков. Так как размер страницы составляет 4096 байт, то номер виртуальной страницы можно получить по текущему счётчику инструкций, сместив его на $\log_2 4096 = 12$ бит вправо, а номер физической страницы из блока трансляции. Тогда проверка может проводиться по Алгоритму 1.

Также необходимо как-то отражать отсутствие записей в *iTLB*, например, выбрав для них некоторое некорректное значение. Учитывая, что гостевой архитектурой является AArch64 с 39-разрядным Linux, биты с 63-го по 39-й всегда равны нулю для адресов

пользовательского пространства и единице для адресов ядра [5], то есть чередующихся бит в этом диапазоне быть не должно, чем можно воспользоваться. Поместим их, например, на позиции 41 и 40, получив 0x0000020000000000. Но, так как сравниваются номера страниц, то нужно также сместить значение вправо на 12 бит: 0x0000000020000000.

```
Вход: GPC, TB.
Начало
    VPN := GPC >> 12
    Index := Hash(VPN)
    Entry := ITLB[Index]
    if VPN != Entry.VPN or TB.PPN != Entry.PPN:
        goto SlowPath
Конец
```

Алгоритм 1. Проверка межстраничного перехода по iTLB.
Algorithm 1. Cross-page jump validation via iTLB.

Алгоритм 1 предполагает четыре сценария:

1. Если запись о странице отсутствует, то сравнение виртуальных номеров провалится, так как запись содержит некорректное для номера страницы значение;
2. может произойти коллизия хэш-функции, тогда сравнение виртуальных номеров также провалится;
3. если текущей гостевой виртуальной странице поставлена в соответствие другая физическая страница, то провалится сравнение физических номеров;
4. если не произошло коллизии, и текущей виртуальной странице соответствует физическая страница, на которой расположена цель перехода, то оба сравнения пройдут успешно.

Для отслеживания актуального виртуального адресного пространства необходимо своевременно обновлять записи в *iTLB*. Так как интерес заключается лишь в исполняемых страницах, то новые записи можно вносить после успешного поиска и генерации блоков трансляции. А в операциях по очистке *iTLB* можно ориентироваться на оригинальный *TLB*.

Далее возникает вопрос о расположении проверочного кода. Есть два существующих решения: генерация кода в начале каждого блока и генерация кода непосредственно перед инструкцией межстраничного перехода. Очевидно, что второй подход будет эффективнее первого, ведь не каждый блок является целью межстраничного перехода. Но для оптимизаций вроде программного предсказания, где появляются целые цепочки прямых переходов, это значительно усложнит генерацию кода, а также процесс его модификации при связывании блоков.

Было разработано следующее решение. Перед началом трансляции и исполнения кода, генерируется шаблон проверочного кода для вставки в начало блока. При генерации каждого блока трансляции перед его целевым кодом выделяется место для проверочного кода. И лишь при необходимости, когда блок становится целью межстраничного перехода, выделенное место заполняется проверочным кодом из шаблона. После чего, для него остаётся только обновить номер физической страницы. Таким образом, проверочный код исполняется только при необходимости, как показано на рис. 2. При связывании блоков, расположенных на одной странице, к адресу перехода добавляется смещение равное размеру проверочного кода. А при межстраничном связывании блоков адресом перехода выступает самое начало блока с проверочным кодом. При исполнении блока трансляции из главного цикла к адресу целевого кода также добавляется смещение.

Недостатком данного решения являются дополнительные затраты памяти, а преимуществом то, что после первого межстраничного связывания, все последующие для того же самого блока-цели не требуют дополнительных операций, в отличии от ленивого подхода, где пришлось бы снова генерировать проверочный код.

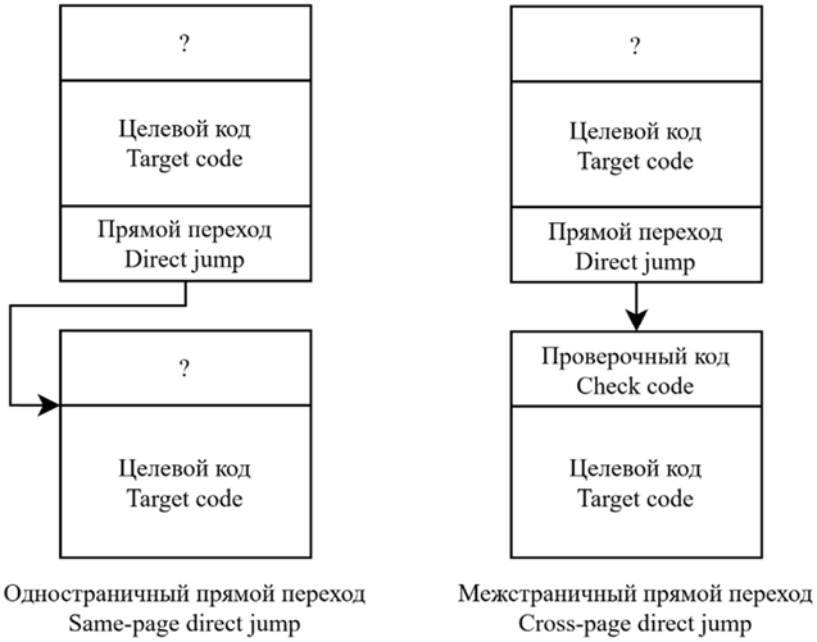


Рис. 2. Связывание блоков с проверочным кодом.
Fig. 2. Block linking with check code.

3.2 Кодирование записей iTLB

На самом деле, для проверки межстраничных переходов можно использовать оригинальную *TLB*, однако введение отдельного буфера позволяет оптимизировать проверочный код. Дабы уменьшить количество обращений к памяти и условных переходов в проверочном коде, номер виртуальной страницы и физической можно кодировать одним 64-разрядным значением. Виртуальный адрес занимает 39 бит, ещё 1 бит нужен, чтобы различать пользовательское пространство и пространство ядра [5], тогда номер виртуальной страницы занимает всего 28 бит. Размер физического адреса не превышает 48 разрядов [6], то есть номер физической страницы занимает не более 36 бит. Суммарно получается ровно 64 бита. Тогда проверка может проводиться по Алгоритму 2. Он включает в себя лишь одно чтение из *iTLB* и один условный переход, что приблизило его к предложенному в работе [3], но с сохранением полной работоспособности.

Важно отметить, так как теперь в записях *iTLB* не хранятся лишние старшие биты, не получится среди них расположить чередующиеся биты для некорректного значения. Однако есть альтернативное решение: в адресном пространстве Linux имеются защитные регионы, например, виртуальная страница с номером 0xffffffffffff входит в один из таких, что позволяет использовать этот номер в качестве некорректного значения для номера виртуальной страницы в *iTLB*, так как на ней точно не будет исполняться код.

3.3 Программное предсказание

Спекулятивное связывание, реализованное в Embra, хотя и улучшает производительность, но ограничивается лишь одним прямым переходом, то есть одной ячейкой предсказания. Помимо этого, усложняется проверочный код, так как добавляется ещё одна точка входа. Было принято решение адаптировать программное предсказание под ограничения полносистемной эмуляции так, чтобы генерация кода была проще, а точность предсказания – выше.

```
Вход: PC, TB.
Начало
    VPN := PC >> 12
    Index := Hash(VPN)
    Value := VPN | (TB.PPN << 28)
    if Value != ITLB[Index]:
        goto SlowPath
Конец
```

Алгоритм 2. Проверка межстраничного перехода по iTLB с закодированными записями.
Algorithm 2. Cross-page jump validation via iTLB with encoded entries.

Программное предсказание заключается во вставке перед вызовом обработчика цепочки предсказаний, каждое из которых состоит из двух частей: сравнение счётчика инструкций с виртуальным адресом цели и условный переход. Таким образом, улучшение производительности зависит от длины цепочки предсказаний и от способа занесения целей в цепочку.

Длина цепочки предсказания была выбрана эмпирическим путём. Полученный график зависимости точности предсказаний от длины цепочки представлен на рис. 3. Под точностью понимается вероятность успешного предсказания. Замеры проводились при загрузке операционной системы, так как она имеет сложные для предсказания переходы, что делает изменения в точности наглядными.

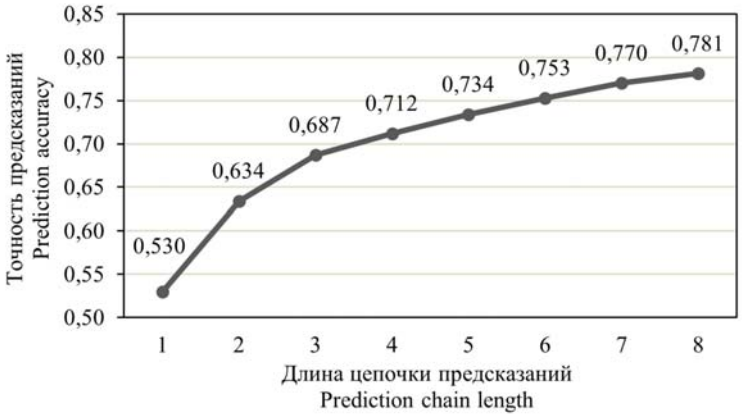


Рис. 3. График зависимости точности предсказаний от длины цепочки предсказаний при загрузке системы.
Fig. 3. Graph of prediction accuracy versus prediction chain length during system boot.

Наиболее заметно точность предсказаний выросла при длине 2 и 3. С учётом того, что увеличение длины цепочки предсказаний значительно увеличивает среднюю стоимость

предсказания, и в QEMU уже имеется поддержка двух прямых переходов для блока трансляции, ограничим длину цепочки предсказаний двумя целями.

Цели предсказаний будут заноситься в цепочку по принципу первого встречного. Этот способ построения цепочек предсказаний является наиболее простым и позволяет использовать стандартный механизм связывания блоков QEMU, за исключением того, что помимо инструкции перехода нужно будет также обновлять инструкцию сравнения.

Цепочка предсказаний может включать в себя цели, расположенные на других страницах, и для них должна проводиться проверка по *iTLB*. Однако есть один нюанс: на самом деле необходимо выполнять проверки по *iTLB* для всех целей, даже если на момент заноса они находятся на той же странице, что и косвенный переход. Связано это с тем, что в QEMU блок трансляции может иметь разные виртуальные адреса, в зависимости от текущего виртуального адресного пространства, а цели предсказаний представлены конкретными виртуальными адресами. Таким образом, даже если на момент заноса в цепочку предсказаний цель была на одной странице с самим блоком, нет никаких гарантий, что в другом виртуальном адресном пространстве это не изменится.

Заметим также, что, хотя проверка цели включает в себя всего две операции (сравнение и условный переход), в две инструкции для выбранной пары архитектур не уложиться без ухищрений. Гостевой счётчик инструкций является 64-разрядным, а инструкция сравнения в x86-64 поддерживает лишь 32-разрядные константы, то есть, чтобы провести сравнение, сначала придётся загрузить адрес цели в регистр. Однако, так как для всех целей предсказаний проводится проверка по *iTLB*, достаточно лишь удостовериться в равенстве младших 12 бит, а старшие биты, то есть номер страницы, будут проверены по *iTLB*. Но, чтобы было меньше промахов, будем сравнивать максимум разрядов, то есть 32.

3.4 Обновление целей программного предсказания

При проведении подсчётов точности предсказаний на тесте производительности CoreMark [7] была замечена определённая закономерность. CoreMark по умолчанию запускается дважды, и второй запуск всегда сопровождался резким падением точности предсказаний, а, следовательно, и производительности. Причиной этому является смена виртуального адресного пространства. Так как QEMU после переключения контекста сохраняет блоки трансляции, то при втором запуске многие блоки трансляции использовались повторно, но с другими виртуальными адресами, а вот адреса целей предсказаний не обновлялись, что и приводило к большому числу промахов.

Для решения данной проблемы можно реализовать механизм обновления целей программного предсказания. Но возникает вопрос, когда проводить обновление. В работе [8] рассматриваются различные типы счётчиков промахов: глобальный счётчик промахов, счётчики промахов для каждой цепочки предсказаний, счётчики промахов для отдельных целей предсказаний и другие. Когда счётчик достигает определённого значения, производится локальная или глобальная очистка, в зависимости от типа счётчика. Наиболее производительными оказались первые два, но глобальный счётчик вышел лишь немного лучше отдельных счётчиков для каждой цепочки и это при том, что он был аппаратным. Тогда будем проводить программное предсказание по Алгоритму 3.

4. Анализ производительности

Чтобы определить изменения в производительности, полученные реализованными оптимизациями, были проведены тесты. Было выбрано несколько сценариев: загрузка системы, сжатие, распаковка, компиляция набора тестов NBench [9] и его исполнение. Результаты тестов были нормированы относительно эталонной производительности, то есть производительности оригинального QEMU 9.1. Таким образом, полученная производительность будет приводится в процентах относительно эталона. Результаты

больше 100% означают повышение производительности, меньше – её снижение. Характеристики машин, на которых производилось тестирование приведены в табл. 2.

```
Вход: GPC.
Начало
    if GPC = GPC1:
        goto HPC1
    if GPC = GPC2:
        goto HPC2
    ChainMissCounter := ChainMissCounter + 1
    if ChainMissCounter = Threshold:
        ClearChain()
        ChainMissCounter := 0
    goto SlowPath
Конец
```

Алгоритм 3. Программное предсказание косвенного перехода.
Algorithm 3. Software prediction of an indirect branch.

Табл. 2. Характеристики машин для тестов производительности.
Table 2. Specifications of the benchmark machines.

Название	Аппаратный стенд	Виртуальная машина
Процессор	i7-8700 @ 3.20GHz	i9-11900KF @ 3.50GHz
RAM	32 ГБ	16 ГБ
Примечание	-	QEMU 8.2.2, SMP 8, KVM

4.1 Межстраничное связывание блоков

Результаты тестирования QEMU с межстраничным связыванием блоков приведены на рис. 4 и рис. 5. В среднем производительность увеличилась на 6,3% для аппаратного стенда и на 12,7% для виртуальной машины, а максимальное улучшение составило 75,3% и 71,4% соответственно. Замедления замечены при загрузке системы и некоторых тестах NBench. Но, согласно рис. 6, они не связаны с количеством межстраничных переходов, например, тест HUFFMAN имеет наибольшее количество переходов, однако на аппаратном стенде было замечено его замедление. Замедления могут быть вызваны тем, что инициализированные межстраничные переходы редко заново использовались, то есть были произведены затраты на вставку проверочного кода, но переход больше не осуществлялся.

4.2 Программное предсказание

Результаты тестирования QEMU с межстраничным связыванием и программным предсказанием приведены на рис. 7 и рис. 8. В среднем производительность увеличилась на 23,5% для аппаратного стенда и на 21,1% для виртуальной машины, а максимальное улучшение составило 89,9% и 76,9% соответственно. При этом замедления, вызванные межстраничным связыванием, были нивелированы программным предсказанием.

Повышение производительности связано с точностью предсказаний, количеством совершённых косвенных переходов и количеством обновлений целей предсказаний. По рис. 9 видно, что все тесты имеют высокую точность предсказаний, в том числе благодаря

механизму обновления целей предсказаний. А как видно по рис. 10 и рис. 11, число обновлений на всех тестах значительно меньше количества совершённых косвенных переходов.

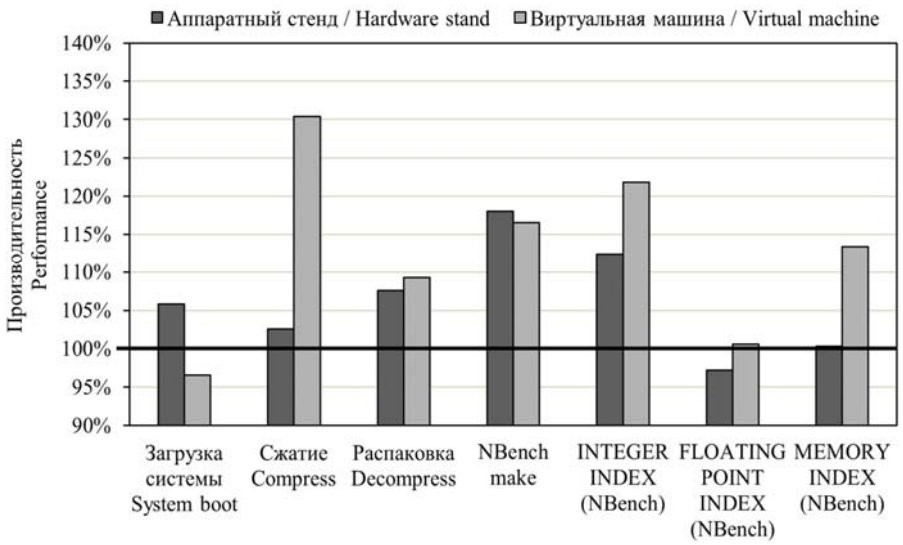


Рис. 4. Производительность, полученная межстраничным связыванием.
Fig. 4. Performance achieved via cross-page linking.

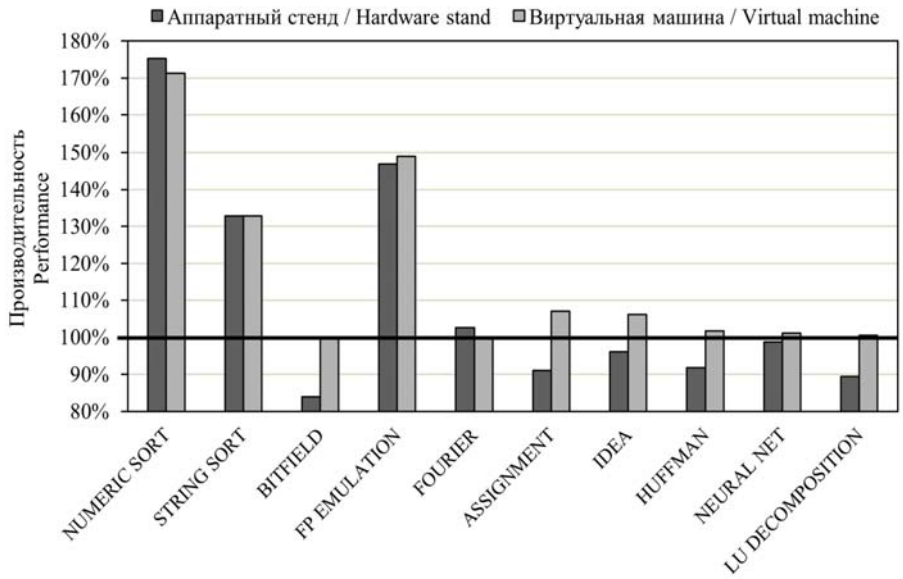


Рис. 5. Производительность, полученная межстраничным связыванием (NBench).
Fig. 5. Performance achieved via cross-page linking (NBench).

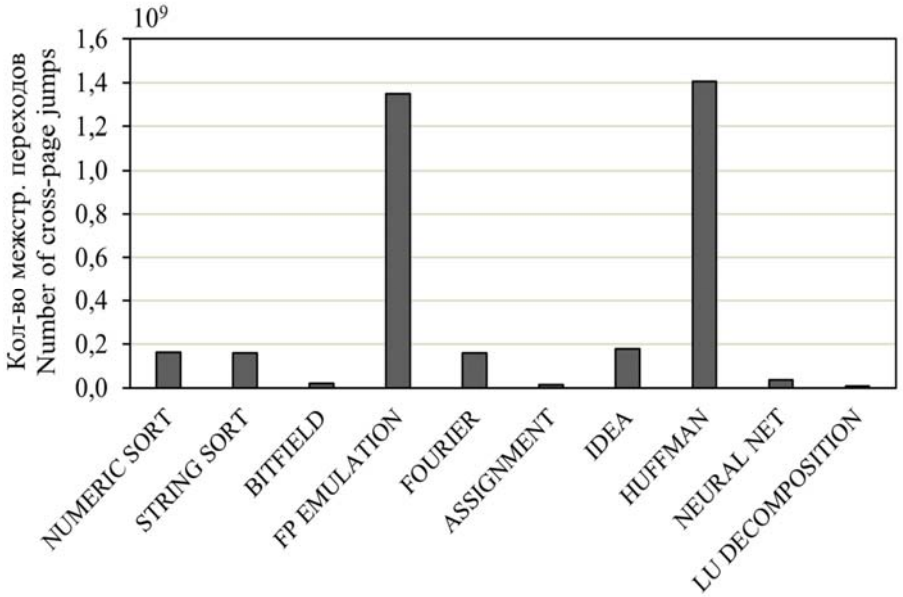


Рис. 6. Количество совершённых межстраничных переходов (NBench).
Fig. 6. Number of executed cross-page jumps (NBench).

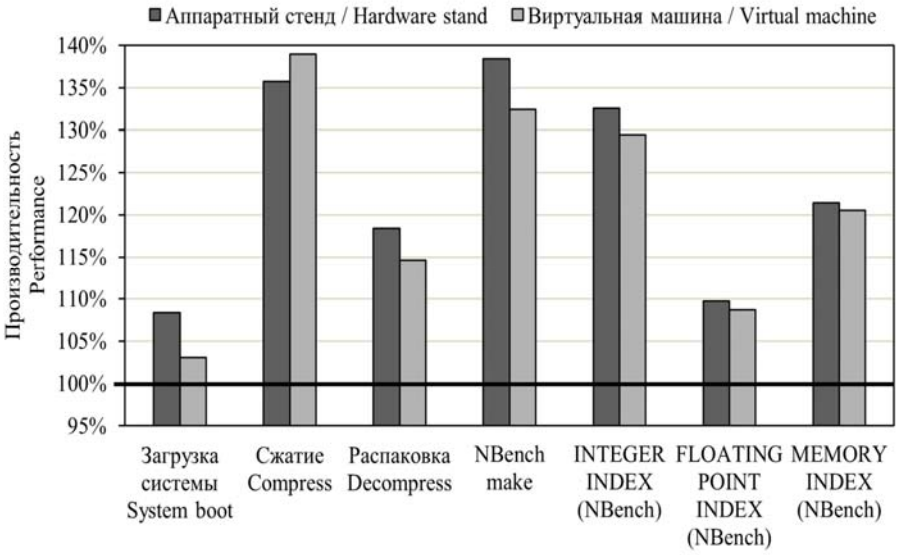


Рис. 7. Производительность,
полученная межстраничным связыванием и программным предсказанием.
Fig. 7. Performance achieved via cross-page linking and software prediction.

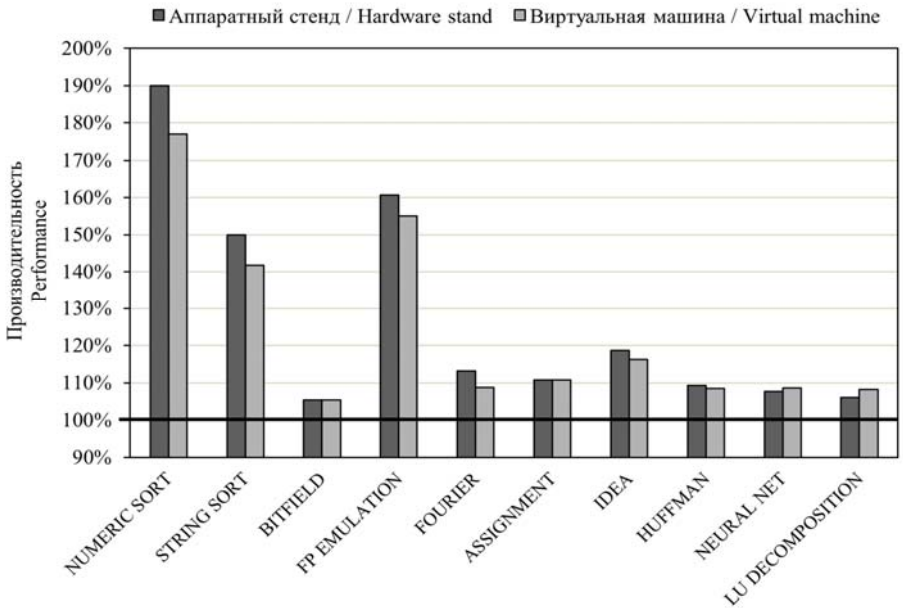


Рис. 8. Производительность,
полученная межстраничным связыванием и программным предсказанием (NBench).
Fig. 8. Performance achieved via cross-page linking and software prediction (NBench).

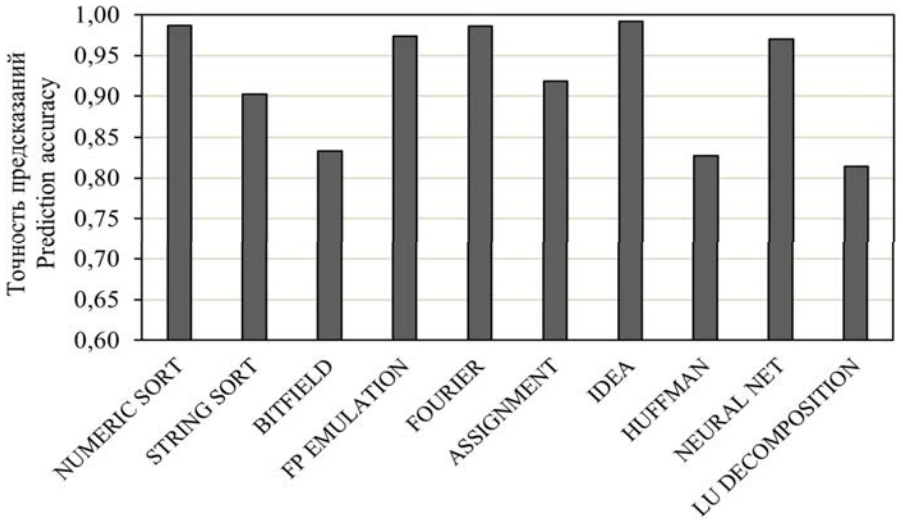


Рис. 9. Точность предсказаний (NBench).
Fig. 9. Prediction accuracy (NBench).

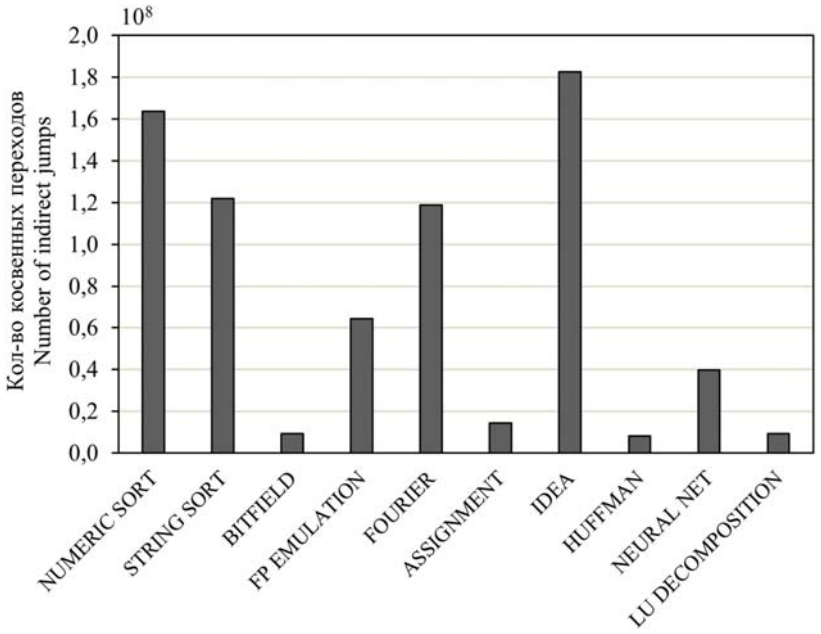


Рис. 10. Количество совершённых косвенных переходов (NBench).
Fig. 10. Number of executed indirect jumps (NBench).

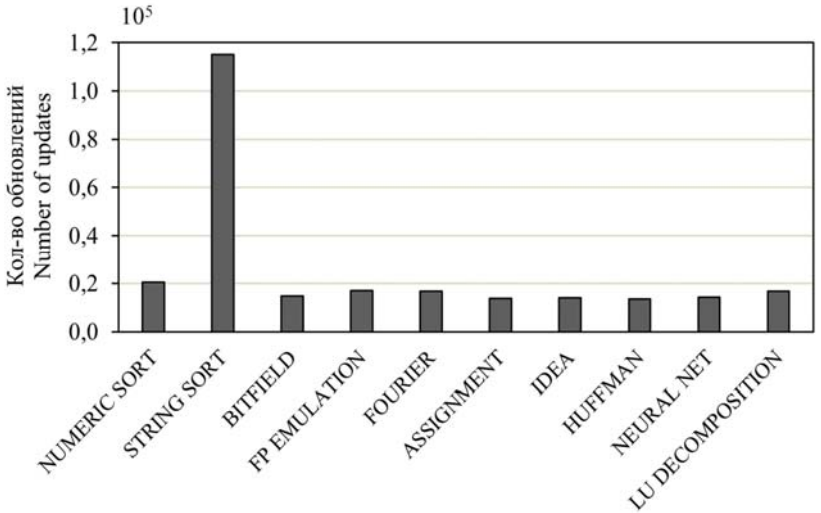


Рис. 11. Количество обновлений цепочек предсказаний (NBench).
Fig. 11. Number of prediction chain updates (NBench).

4.3 Увеличение затрат памяти

Было выяснено, что при загрузке системы в добавлении проверочного кода нуждаются 25,8% блоков трансляции, а затраты памяти увеличились на 15,9% относительно оригинального QEMU, но в сравнении с ленивой проверкой межстраничных переходов лишь на 4,9%. Такое маленькое увеличение затрат памяти относительно ленивой проверки объясняется тем, что в её случае проверочный код нужно добавлять для каждой ячейки предсказаний, то есть по две штуки на один блок с косвенным переходом, а в случае представленного решения по одной на блок, правда на каждый. Но, чем больше косвенных переходов, тем меньше увеличение затрат на память, а обычно они встречаются довольно часто. Такая цена является приемлемой для упрощения генерации кода.

5. Заключение

В статье были рассмотрены основные оптимизации передачи управления в условиях динамической двоичной трансляции. Для полносистемной эмуляции, как наиболее подходящие, были выбраны межстраничное связывание и программное предсказание.

В статье было доказано, что один из существующих методов проверки межстраничных переходов неверен, а для корректного был улучшен алгоритм проверки с учётом особенностей гостевой архитектуры. Помимо этого, в статье был представлен новый подход к реализации программного предсказания в полносистемных эмуляторах, который упрощает проверочный код и увеличивает точность предсказания.

Данные оптимизации были реализованы в полносистемном эмуляторе QEMU версии 9.1, а после проведены тесты производительности, которые показали значительное улучшение относительно оригинального QEMU.

Список литературы / References

[1]. Bellard F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, 2005, pp. 41-46.
[2]. Witchel E., Rosenblum M. Embra: fast and flexible machine simulation. In Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '96), 1996, vol. 24, no. 1, pp. 68-79. DOI: 10.1145/233013.233025.
[3]. Hong D.-Y., Hsu C.-C., Chou C.-Y., Hsu W.-C., Liu P., Wu J.-J. Optimizing Control Transfer and Memory Virtualization in Full System Emulators. ACM Transactions on Architecture and Code Optimization, 2015, vol. 12, no. 4, Article 47, pp. 1-24. DOI: 10.1145/2837027.
[4]. D'Antras A., Gorgovan C., Garside J.D., Luján M. Optimizing Indirect Branches in Dynamic Binary Translators. ACM Transactions on Architecture and Code Optimization, 2016, vol. 13, no. 1, Article 7, pp. 1-25. DOI: 10.1145/2866573.
[5]. Memory Layout on AArch64 Linux – The Linux Kernel documentation. Available at: https://www.kernel.org/doc/html/v5.3/arm64/memory.html, accessed 31.10.2025.
[6]. ARM Ltd. ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). 2013. 5158 p.
[7]. CoreMark. Available at: https://github.com/eembc/coremark, accessed 31.10.2025.
[8]. Jia N., Yang C., He Y., Cheng X. SPTU: Improving Dynamic Binary Translation through Software Prediction with Target Updating. In Proceedings of International Conference on Systems and Storage (SYSTOR 2014), 2014, pp. 1-12. DOI: 10.1145/2611354.2611368.
[9]. Linux/Unix NBench. Available at: https://github.com/toshsan/nbench, accessed 31.10.2025.

Информация об авторах / Information about authors

Максим Алексеевич КОСТИН – инженер отдела компиляторных технологий ИСП РАН. Сфера научных интересов: эмуляторы, динамическая двоичная трансляция, оптимизации.

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

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.

Дмитрий Николаевич ПОЛЕТАЕВ – разработчик ПО, 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.

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.

Наталья Игоревна ФУРСОВА – кандидат технических наук, старший научный сотрудник, старший преподаватель. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, эмуляторы.

Natalia Igorevna FURSOVA – Cand. Sci. (Tech.), senior researcher, senior lecturer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, emulators.