

С.С. Гилязов, Е.М. Кравцунов, П.В. Пантелеев

Salavat Giliyazov, Evgeny Kravtsunov, Pavel Panteleev

**ОПТИМИЗАЦИЯ ЯДРА ОС LINUX ДЛЯ АРХИТЕКТУРЫ «ЭЛЬБРУС» С  
ПОДДЕРЖКОЙ NUMA**

**A SET OF LINUX KERNEL OPTIMIZATIONS FOR «ELBRUS» ARCHITECTURE  
WITH NUMA SUPPORT**

*Описаны особенности поддержки NUMA в ядре ОС Linux для архитектуры «Эльбрус». Рассматриваются изменения в архитектурно-независимой и архитектурно-зависимой частях ядра, связанные с NUMA и реализацией библиотеки libnuma. Перечислены разработанные авторами методы оптимизации подсистемы памяти ядра, эффективные для NUMA платформ. Описан алгоритм оптимизации, суть которой – создание копий исполняемого кода и константных данных ядра на каждом узле NUMA машины с целью минимизации времени вызова функций ядра на всех узлах. Реализация алгоритма позволила уменьшить среднее время активизации процесса на 40% для машин, имеющих архитектуру «Эльбрус» с поддержкой NUMA. Результаты измерений производительности получены на машине Эльбрус-3S, имеющей четыре узла NUMA.*

*Keywords: Linux Kernel performance, Elbrus, NUMA, per-node copies of kernel image, use huge pages for kernel*

**Введение**

NUMA (Non Uniform Memory Access) – это способ организации взаимодействия процессоров с памятью, при котором процессоры группируются в объединяемые быстрыми каналами узлы, связанные с определенными областями общей памяти. Каждый процессор имеет доступ ко всем областям памяти, хотя времена доступа к памяти

«своего» и «чужих» узлов заметно отличаются. Такой системе принципиально свойственна масштабируемость, ибо число узлов, а, следовательно, и объем разделяемой памяти могут многократно расширяться. В этом и состоит основное достоинство NUMA-платформ перед системами класса SMP (Symmetric Memory Processing), объединяющими процессоры на единой памяти через общую шину или коммутатор, – их возможности масштабирования существенно скромнее.

Многопроцессорный вычислительный комплекс Эльбрус-3S включает в себя четыре узла NUMA, в каждый из которых входит один процессор Эльбрус-S<sup>1</sup> со своей областью общей памяти, подключенной через отдельную шину. Узлы комплекса связаны межпроцессорными линками.

Вычислительный комплекс Эльбрус-3S работает под управлением ОС Linux с ядром 2.6.14, портированным на архитектуру «Эльбрус». Поддержка NUMA в ядре ОС Linux реализована как в архитектурно-независимой, так и в архитектурно-зависимой частях. Логически все изменения в ядре можно разделить на две группы, одна из которых связана с поддержкой библиотеки libnuma, позволяющей управлять политиками выделения памяти для пользовательских задач, другая – с оптимизациями ядра ОС Linux для архитектуры «Эльбрус», предназначенными для ускорения его работы.

## **1. Политики выделения памяти. Поддержка библиотеки libnuma в ядре ОС Linux**

Средняя производительность комплекса существенно зависит от природы исполняемых задач, которые в этом смысле можно разделить на три типа:

1) задачи, чувствительные к задержкам (latency), которые связаны с обращениями в память;

---

<sup>1</sup> В связи с этим в дальнейшем можно было бы исключить термин «узел», заменив его термином «процессор». Но, т.к. описанные в статье программные оптимизации универсальны и применимы к любым топологиям NUMA с процессорами архитектуры «Эльбрус», авторы решили и далее использовать понятие «узел».

- 2) задачи, чувствительные к полосе пропускания (bandwidth);
- 3) задачи смешанного типа.

Задачи первого типа, как правило, плохо распараллеливаются по памяти. Для повышения производительности в этом случае необходимо выделять память в том же узле, где выполняется задача.

Задачи второго типа, напротив, хорошо распараллеливаются по памяти. Примером является параллельная запись в память большого количества блоков данных. Для таких задач решающим, с точки зрения производительности, является полоса пропускания – количество процессоров, совместно выполняющих эту работу. Если процессоров много, то задержка доступа в память, связанная с выделением страниц на чужом узле, не оказывает существенного влияния на среднюю производительность.

Задачи смешанного типа могут быть чувствительны и к задержкам, и к полосе пропускания, в зависимости от стадии выполнения. Пиковая производительность для них может быть достигнута с использованием адаптивных алгоритмов миграции страниц памяти [3, 4, 5].

В ядре ОС Linux 2.6.14 поддерживаются четыре политики выделения памяти для NUMA:

**default** (политика по умолчанию) – выделение памяти в локальном узле, где выполняется задача;

**bind** – выделение памяти в заданном наборе узлов. Попытка выделения памяти в узле, не входящем в набор, приводит к ошибке;

**interleave** – распределение выделяемых страниц памяти по всем имеющимся узлам равномерно;

**preferred** – выделение памяти преимущественно в локальном узле. Если там память исчерпана, то она выделяется в другом узле. В этом качестве выбирается соседний (в соответствии с топологией) узел. Если памяти нет и в соседнем узле, то

предпринимается попытка выделить память в узле, ближайшем к соседнему. Процесс заканчивается успешным выделением памяти в одном из узлов или ошибкой, если память исчерпана во всех узлах.

Политики выделения памяти могут быть связаны одной из трех сущностей:

- задачей (процессом);
- мапированной анонимной областью памяти (областью памяти, выделенной для процесса с помощью системного вызова `mmap` с флагом `MAP_ANON`);
- областью разделяемой памяти (памяти, полученной с помощью системного вызова `shmat` или путем мапирования файлов с файловых систем `tmpfs`, `shmfs`).

Политики, установленные для мапированных анонимных областей памяти, обладают бóльшим приоритетом, чем политики, установленные для задачи.

Для установки политик в ядре ОС Linux 2.6.14 введен тип `nodemask_t` и реализованы следующие системные вызовы:

**set\_mempolicy** – системный вызов для установки политики выделения памяти для задачи, а также для областей разделяемой памяти;

**mbind** – системный вызов для установки политики выделения памяти для мапированных анонимных областей памяти;

**get\_mempolicy** – системный вызов, возвращающий значение политики выделения памяти для задачи или области разделяемой памяти.

Для политик выделения памяти существует библиотека `libnuma` [1], предоставляющая набор интерфейсов управления политиками выделения памяти, используемых с утилитой `numactl`. Поддержка библиотеки `libnuma` в ядре 2.6.14 заключается в реализации системных вызовов `set_mempolicy`, `get_mempolicy`, `mbind` и поддержке специального набора файлов в файловой системе `sysfs`, обеспечивающего хранение и доступ к данным о топологии узлов и областей памяти в NUMA системе. Портирование этой поддержки на архитектуру «Эльбрус» не было сложной задачей и

выполнялось авторами по аналогии с тем, как это сделано в ядре для других архитектур, на основе которых строятся NUMA-платформы (ia64 – Intel, x86\_64 – AMD).

Использование `libnuma` позволяет улучшить производительность приложений, работающих на Эльбрус-3S. Однако, для получения пиковой производительности вычислительного комплекса стандартных средств, каким является `libnuma`, оказывается недостаточно. Производительность, близкая к максимальной, достигается путем ряда оптимизаций ядра, основанных на особенностях архитектуры «Эльбрус».

## 2. Оптимизации архитектурно-зависимой части ядра ОС Linux

С целью эффективной реализации поддержки NUMA в ядре авторами выполнены следующие оптимизации:

- 1) использование больших страниц для исполняемого кода ядра;
- 2) использование глобального контекста для виртуального пространства ядра;
- 3) создание копий исполняемого кода ядра в памяти каждого узла NUMA системы;
- 4) расположение области для переменных и объектов (регсри области) в памяти сформировавшего ее узла;
- 5) для потоков ядра, создаваемых механизмом `kthread` с привязкой к конкретному узлу, – возможность заказа всех ресурсов в памяти узла;
- 6) выделение буферов для работы с DMA на каждом узле.

Выигрыш по производительности, измеренный и описанный в статье, получен за счет оптимизаций 1, 2, 3 (эффект от оптимизаций 4, 5, 6 не проверялся измерениями – это может быть темой отдельной работы, т.к. такой эксперимент весьма сложен, с точки зрения моделирования потока загрузки ОС при активном использовании DMA устройств). Для их характеристики необходимо в общих чертах рассмотреть алгоритм трансляции виртуальных адресов в физические, реализованный в архитектуре «Эльбрус», в частности, в машине Эльбрус-3S.

Трансляция виртуального адреса в физический осуществляется аппаратно – с использованием устройства TLB (Translation Lookaside Buffer) и аппаратной поддержки поиска по таблице страниц. Как и в большинстве архитектур, в данной используется четырехуровневая таблица страниц, описание которой можно найти, например, в [6].

При поступлении в процессор запроса на трансляцию виртуального адреса в физический выполняется поиск в таблице TLB, позволяющей найти физический адрес, зная виртуальный адрес и контекст процесса. Контекст процесса – это число, определяющее конкретный процесс, для которого должна быть выполнена трансляция. Поиск по TLB является быстрым, но размер TLB ограничен – его общая емкость составляет 1024 строки. Если искомый физический адрес найден, то трансляция выполняется быстро; такое событие называется «hit». Если результат не успешен («промах» или «miss»), то дальнейший поиск производится аппаратно по таблице страниц процесса. Если он завершился успешно, то найденный адрес аппаратно заносится в строку TLB, после чего поиск по TLB запускается снова – в этом случае он даст результат hit.

Таблица страниц заполняется из ядра. Добавление строк в таблицу производится при обработке исключительной ситуации `page_miss`. Она возникает, если в процессе трансляции искомый адрес не найден ни в TLB, ни в таблице страниц. Обработчик в ядре выделяет страницу памяти, в которую было обращение, записывает строку в таблицу страниц и обнуляет TLB. После возврата из обработчика `page_miss` трансляция запускается повторно и завершается тем, что искомый адрес обнаруживается в таблице страниц, найденный адрес заносится в TLB.

Для дальнейшего изложения важно отметить, что в реализации операционной системы Linux, применительно к архитектуре «Эльбрус», ее страницы, содержащие исполняемый код ядра, не откачиваются из памяти, т.е. постоянно являются резидентами в основной памяти, обращения по адресам этих страниц не приводят к исключительным ситуациям `page_miss`, а трансляция соответствующих виртуальных адресов завершается

успешно – нахождением адреса или в TLB, или (что займет больше времени) в таблице страниц.

#### *Использование больших страниц для исполняемого кода ядра*

Из приведенного описания следует, что время трансляции виртуального адреса в физический является минимальным, если строка с искомым адресом находится в TLB (ситуация hit). Архитектура «Эльбрус» поддерживает на аппаратном уровне работу со страницами двух размеров: обычные страницы размером 4 Кбайт и большие страницы размером 4 Мбайт. Обычные страницы используются при мапировании физической памяти для приложений и процессов пользователя и ядра, а также для мапирования аппаратных и программных стеков. Большие страницы используются для хранения исполняемых кодов ядра и модулей. В этом случае большие страницы позволяют сэкономить свободные строки в TLB за счет того, что весь исполняемый код ядра помещается в несколько (от 4 до 8) больших страниц, которым соответствует лишь несколько строк. Сэкономленные строки TLB могут использоваться для хранения большего числа адресов обычных страниц, а это повышает производительность машины в целом, т.е. операционной системы с работающими под ней приложениями.

#### *Использование глобального контекста для адресов страниц ядра*

Поиск в TLB завершается успешно, если для найденного физического адреса виртуальный адрес и значение контекста совпадают с заданными в запросе на трансляцию. В архитектуре «Эльбрус» реализована аппаратная поддержка глобального контекста, при которой значение контекста не участвует в сравнении. В целях повышения производительности системы в целом в ядре также реализована следующая оптимизация: все страницы ядра относятся к глобальному контексту, и при занесении в таблицу страниц строки, описывающей страницу ядра, для этой строки устанавливается признак

глобального контекста. Эта оптимизация позволяет избежать промахов в TLB по страницам ядра при переключении между процессами с разными контекстами, поскольку страницы ядра являются глобальными для всех процессов.

Описанные оптимизации являются эффективными для любых систем. Для NUMA системы их эффективность может быть еще повышена за счет оптимизации путем создания копий исполняемого кода ядра в памяти каждого узла NUMA. Напомним, что код ядра загружен в большие страницы и имеет глобальный контекст.

### **3. Алгоритм создания копий ядра на узлах Эльбрус-3S**

Оптимизация возможна благодаря следующей особенности реализации ядра ОС Linux применительно к архитектуре «Эльбрус»: образ ядра загружается в собственное виртуальное пространство, т.е. для кода, модулей и данных ядра, доступных только по чтению, выделен специальный диапазон виртуальных адресов. В ОС Linux все виртуальное пространство разделено на две части: виртуальные адреса со значениями меньше `TASK_SIZE` могут использоваться в режиме пользователя, виртуальные адреса со значениями выше `TASK_SIZE` предназначены для ядра. В варианте архитектуры «Эльбрус» виртуальное пространство, предназначенное для ядра, также разделено на пять непересекающихся диапазонов адресов: 1) диапазон для мапирования физической памяти, 2) диапазон для мапирования образа ядра и модулей, 3) диапазон для мапирования аппаратных и пользовательских стеков, 4) диапазон для мапирования пространств ввода-вывода, 5) диапазон для мапирования таблиц страниц. Диапазон для мапирования образа ядра и модулей начинается со значения адреса `VM_KERNEL_BASE` и имеет размер `kernel_image_size = "text" + "nodedata" + "data" + "bss"`, где размеры областей `text`, `nodedata`, `data` и `bss` вычисляются из значений соответствующих меток в объектном файле ядра `vmlinux`.

Создание копий производится для страниц исполняемого кода и данных ядра,

доступных только по чтению, адрес которых принадлежит диапазону виртуальных адресов от `VM_KERNEL_BASE` до `KERNEL_NODEDATA_END`, где `KERNEL_NODEDATA_END` – последний адрес сегмента `nodedata` (или начальный адрес сегмента `data`). Копирование производится при начальной загрузке ядра после получения ядром управления от программы начальной загрузки (`bootloader`) и до переключения на виртуальную адресацию. Копирование производится по физическим адресам, с узла, которому принадлежит ведущий процессор (`bootstrap`), на остальные узлы. После копирования и до переключения на работу по виртуальным адресам на всех узлах создаются копии таблиц страниц. Эти таблицы после переключения на виртуальную адресацию используются для трансляции адресов. После построения таблиц для любого адреса из диапазона `[VM_KERNEL_BASE; KERNEL_NODEDATA_END]` на каждом узле NUMA системы однозначно определен физический адрес. Ясно, что после создания копий физические адреса, соответствующие одному и тому же виртуальному адресу из указанного диапазона, различаются. После успешного завершения построения таблиц страниц на всех узлах происходит переключение на виртуальную адресацию, после чего на `bootstrap` процессоре вызывается архитектурно-независимая функция `start_kernel()`.

Важно отметить, что в течение промежутка времени – с момента передачи ядру управления от программы `bootloader` до момента начала копирования по физическим адресам – аппаратная инициализация всех процессоров выполняется параллельно. Это важная особенность реализации ядра ОС Linux для архитектуры «Эльбрус», которая при масштабировании системы по количеству процессоров дает явный выигрыш во времени старта системы по сравнению с традиционным подходом, когда управление от `bootloader` передается не всем имеющимся процессорам одновременно, а только `bootstrap` процессору.

Три описанные оптимизации в совокупности позволили получить существенный прирост производительности ядра.

#### 4. Измерение среднего времени активизации процесса

Измерение производилось с помощью теста `rt_model`, который имитирует работу системы реального времени. Источником прерывания в тесте является контроллер прерываний LAPIC [2]. Тест моделирует работу двух потоков: первый поток ожидает прерывание, выполняет некоторую полезную работу (вычисление), после чего активизирует второй поток, который выполняет некоторые вычисления и переходит в состояние ожидания активизации.

Тест измеряет максимальное время активизации процесса. Измерения производились на опытном образце машины Эльбрус-3S, имеющем четыре узла NUMA – по одному процессору в каждом узле. Характеристики процессора: тактовая частота – 500 МГц, L1-кэш – 64 Кбайт, L2-кэш – 2048 Кбайт, `bogomips` = 917. Результаты измерений приведены в табл. 1.

Таблица 1

	<b>NUMA</b>	<b>NUMA.optim</b>
Время активизации процесса (мкс)	34	20

#### 5. Другие возможности оптимизации

В рамках исследования способов повышения производительности ОС на NUMA машинах авторами, помимо основных методов оптимизации на уровне управления памятью, был предложен и реализован ряд методов, которые могут быть эффективны для модулей ядра, использующих регистры переменные, и при работе с DMA устройствами.

##### *Расположение регистры области в памяти сформировавшего ее узла*

В ядре Linux 2.6.14 области для хранения регистры переменных располагаются в памяти последовательно, без учета топологии NUMA машины, т.е. для некоторого узла может получиться так, что его регистры переменные хранятся в памяти другого узла и

обращения к ним чреваты затратами времени. Оптимизация заключается в расположении каждой области регсри в памяти своего узла. К настоящему времени она уже реализована разработчиками Linux-сообщества в последующих версиях ядрах.

*Для потоков ядра, создаваемых механизмом kthread с привязкой к конкретному узлу,  
– реализация возможности заказа всех ресурсов памяти в памяти узла*

В ядре ОС Linux для архитектуры «Эльбрус» имеется определенный набор потоков ядра, формируемых с помощью механизма kthread, которые после создания привязываются к определенному процессору. В рамках оптимизации для таких потоков реализована возможность заказывать все ресурсы в памяти того узла, к которому будет выполняться привязка, а не того, на котором поток создается.

*Выделение буферов для работы с DMA устройствами в каждом узле*

Буферы памяти для работы с DMA устройствами выделяются на каждом узле NUMA машины, причем каждое устройство, как правило, физически привязывается к конкретному узлу. Для сокращения расходов на доступ заказ памяти выполняется из области DMA-памяти соответствующего узла.

Также реализована оптимизация создания копий вспомогательных буферов (bounce буферов) в каждом узле NUMA машины. Bounce буферы предназначены для промежуточного хранения при передаче данных из устройства DMA в оперативную память зоны NORMAL или HIGHMEM [6].

### **Литература**

1. Andy Kleen. An NUMA API for Linux, SUSE Labs, august 2004.
2. Семенихин С.В., Ревякин В.А., Ананьев Л.И., Костин Д.А., Гилязов С.С., Харитонов М.И., Ситников А.В., Чернис Е.Н., Лебедев В.П. ОС Linux и режим реального

времени. – «Вопросы радиоэлектроники», серия ЭВТ, 2009, вып. 3.

3. Lee T. Automatic Page Migration for Linux, Schermerhorn, HP, Linuxconf Sydney, 2007.

4. Christoph Lameter. Bazillions of pages, Linux Symposium Ottawa, july 2008.

5. Brice Goglin, Nathalie Furmento. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux, INRIA, MTAAP'09, 2009.

6. Д. Бовет, М. Чезати. Ядро Linux. СПб., БХВ-Петербург, 2007.