Московский государственный университет имени М.В. Ломоносова Факультет вычислительной математики и кибернетики Кафедра автоматизации систем вычислительных комплексов

Курсовая работа

Реализация программного интерфейса для эффективных статистических счётчиков в ядре операционной системы FreeBSD

Автор:
Г. А. Смирнов,
группа м210

Научный руководитель:
к.ф.-м.н., с.н.с.
Д. Ю. Гамаюнов

Москва 2013

Оглавление

Введение 3

1 Аллокатор per-CPU памяти 4

1.1 Особенности работы с per-CPU памятью 4

1.2 Существующая per-CPU память в ядре FreeBSD 5

1.2.1 Статический pcpu 5

1.2.2 Динамический pcpu 6

1.3 Аллокатор per-CPU памяти в NetBSD 6

1.4 Аллокатор per-CPU памяти в GNU Linux 6

1.5 Реализация per-CPU зон для универсального аллокатора ядерной памяти *uma(9)* в FreeBSD 7

1.5.1 API *uma(9)* 7

1.5.2 Внутреннее устройство *uma(9)* 7

1.5.3 Реализация аллокатора per-CPU памяти в рамках аллокатора *uma(9)* 10

2 per-CPU счётчики 13

2.1 Счётчики в BSD в ретроспективе 13

2.2 Использование per-CPU памяти для реализации счётчиков 14

2.2.1 per-CPU счётчики в GNU Linux 15

2.2.2 API для системных счётчиков FreeBSD 16

2.2.3 Реализация per-CPU счётчиков в FreeBSD с использованием критических секций 17

2.2.4 Обновление счётчика без критической секции 18

2.3 Анализ результатов 20

2.3.1 Сравнение нового интерфейса, обычного инкремента и атомарного инкремента в синтетическом микробенчмарке 20

2.3.2 Сравнение нового интерфейса, обычного инкремента и атомарного инкремента с помощью инструментов аппаратного анализа производительности 22
2.3.3 Использование нового интерфейса в реальных задачах 24
2.4 Дальнейшая работа 24
2.4.1 Применение *counter(9)* в ядре FreeBSD 24
2.4.2 Лёгкий счётчик ссылок на основе *counter(9)* 25

Аннотация

В работе рассматриваются проблема ведения эффективного статистического учёта событий в ядре операционной системы, при условии что события происходят с высокой частотой и на разных процессорах. Демонстрируется реализация аллокатора приватной памяти для процессоров, реализация статистических счётчиков в этой памяти в ядре операционной системы FreeBSD. Проводится сравнение новой реализации с классическими методами, демонстрируется применение в реальных задачах.

Введение и постановка задачи

В операционных системах, работающих на многопроцессорных машинах, задача параллельного доступа процессоров к данным, как правило, решается с помощью средств синхронизации, реализованных с помощью атомарных инструкций. Средства синхронизации, такие как спинлоки, гарантируют консистентность данных в любой момент времени, но достигается это ценой на синхронизацию, как правило, весьма ощутимой. Альтернативой синхронному доступу к общей области памяти является расположение в памяти отдельной приватной копии данных для каждого процессора. Очевидно, что не любая задача может быть решена без синхронного доступа, но в тех случаях, когда это возможно, решение с приватными процессорными данными (*per-CPU data*) позволяет избежать расходов на синхронизацию и добиться повышения производительности.

Типичным примером такой задачи является сбор статистических данных в ядре операционной системы. Параллельные потоки выполнения могут одновременно обновлять значения счётчиков, и это хотелось бы совершать без потери данных и за возможно меньшее количество процессорных тактов.

В данной работе ставятся следующие задачи:

- В рамках ядерного slab аллокатора памяти *uma(9)* в ядре FreeBSD реализовать возможность динамически аллоцировать память в виде per-CPU приватных регионов.

- Разработать и реализовать API для статистических счётчиков, обновление которых было бы thread safe и как можно более быстрым.

Глава 1

Аллокатор per-CPU памяти

1.1 Особенности работы с per-CPU памятью

Одним из важных свойств вытесняющей многозадачности является то, что, тред выполняемый прямо сейчас на процессоре, может быть вытеснен другим тредом, отложен в очередь планировщика, а позже снова запущен на другом процессоре. То есть тред может мигрировать с одного процессора на другой. Если тред ведёт работу с какой-то областью памяти приватной для процессора, и держит на своём стеке исполнения или в регистрах указатели на неё, то после миграции это будут указатели на приватную область памяти другого процессора. Чтобы не допустить такую ситуацию, ещё до получения адреса приватного региона памяти тред должен привязать себя к текущему процессору, то есть указать планировщику, что его невозможно выполнять на иных процессорах. После окончания работы с per-CPU памятью привязка треда устраняется.

Однако, в большинстве случаев и привязки к процессору недостаточно для безопасной работы с приватными данными. Представим следующую последовательность событий: тред А получает указатель на приватную память, и затем тред А осуществляет инкремент переменной в этой памяти. Для этого он сначала читает значение ячейки памяти в регистр, затем инкрементирует регистр и записывает новое значение из регистра в память. Если после чтения данных из памяти тред А будет снят планировщиком с процессора, а на процессоре будет запущен тред Б, который проделает ту же самую последовательность, то потом, когда тред А вернётся на процессор и продолжит исполнение, он перезапишет обновление сделанное тредом Б, и таким образом мы потеряем данные. Получается, необходим запрет вытеснения треда с процессора на всё время выполнения работы с приватной памятью.

Заметим, что это более сильное ограничение, чем привязка к текущему процессору.

Такой запрет вытеснения в ядерном программировании получил название *критической секции*. Тред обозначает своё желание не уходить с процессора *входом* в критическую секцию и окончание критической секции *выходом*. Критические секции могут быть вложенными, тогда для того, чтобы критическая секция завершилась, число выходов должно быть равным числу входов. Не следует путать критические секции в ядре с мьютексами, как это часто бывает в литературе по программированию юзерленд приложений.

Доступ к per-CPU памяти практически во всех случаях требует критической секции.

Следует отметить, что критические секции могут быть "мягкими" и "жёсткими". "Мягкие" критические секции запрещают вытеснение на уровне планировщика задач ядра. То есть прерывания могут прервать выполнение текущего контекста, однако операционная система гарантирует, что после завершение обработчика прерывания, планировщик вернёт исполнение именно текущему треду и ни кому иному. "Жёсткая" критическая секция представляет собой полный запрет прерываний на текущем процессоре, что на большинстве машин достаточно дорогостоящая операция.

1.2 Существующая per-CPU память в ядре FreeBSD

1.2.1 Статический pcpu

При реализации SMP ядра операционной системы обязательно возникает необходимость иметь отдельную структуру данных, описывающую каждый отдельный процессор. В ядре FreeBSD эта структура называется *struct pcpu*. Эта структура содержит как машинно-независимую, так и машинно-зависимую части. Помимо прочих полей в машинно-независимой части расположены: указатель на структуру описывающую текущий выполняемый тред и указатель на структуру описывающую контекст исполнения, обращение к которым происходит весьма часто во время выполнения кода в ядре. В связи с этим на большинстве архитектур адрес приватной *struct pcpu* не вычисляется всякий раз как требуется к нему обратиться, а хранится в постоянном регистре. Реализация хранения этих структур в памяти машинно-зависима. На большинстве архитектур они расположены на bss в виде последовательного массива, размер которого соответствует числу процессоров. Инициализируются структуры на раннем этапе загрузки ядра после определения числа процессоров в машине. Далее по тексту мы будем называть эту память *статическая per-CPU память* или

статический pcpu.

1.2.2 Динамический pcpu

У различных ядерных модулей может возникать необходимость иметь свою per-CPU память для каких-то целей, и для этого существует API *DPCPU*. Подсистема *DPCPU* преаллоцирует небольшой регион памяти для каждого процессора на раннем этапе загрузки. Над этим регионом работает примитивный first-fit extent аллокатор, отрезающий необходимое количество памяти при загрузке модулей, если код модуля содержит данные в специальной ELF секции, именованной *set_pcpu*. API не предусматривает аллокаций из этого региона на этапе исполнения, хотя теоретически это возможно. Из вышесказанного ясно, что применимость *DPCPU* весьма ограничена.

1.3 Аллокатор per-CPU памяти в NetBSD

Ядро NetBSD предоставляет своим модулям API *percpu(9)*, позволяющее аллоцировать per-CPU память во время исполнения. Память запрашивается с помощью функции *percpu_alloc()*, которая возвращает *percpu_t*, который позже с помощью функции *percpu_getref()* может быть преобразован в указатель *void **, указывающий на приватную для текущего процессора аллокацию.

Аллокатор *percpu(9)* реализован в файле *kern/subr_percpu.c* [1]. Реализация такова: для каждого процессора в системе аллоцируется регион памяти. Аллокация отдельных элементов из регионов осуществляется с помощью ресурсного аллокатора общего назначения *vmem(9)*[2], где ресурсом являются смещения в этих регионах.

Важной особенностью является то, что *percpu_getref()* не только вычисляет указатель на память приватную для текущего процессора, но и осуществляет вход в критическую секцию, то есть запрещает вытеснение текущего треда. После доступа к приватной памяти следует вызвать *percpu_putref()*, которая завершает критическую секцию.

1.4 Аллокатор per-CPU памяти в GNU Linux

В ядре Linux также предоставлен API для аллокации per-CPU памяти во время исполнения [3]. Память аллоцируется с помощью *alloc_percpu()*. Доступ к региону, приватному для текущего процессора, осуществляется с помощью макроса

get_cpu_var(). Макрос начинается с *preempt_disable()*, то есть со входа в критическую секцию. После работы с приватной памятью следует вызвать *put_cpu_var()*, которая завершит критическую секцию.

Как видно из краткого обзора, различные операционные системы, такие как NetBSD и GNU Linux, используют аналогичные идеи для реализации поставленной задачи.

1.5 Реализация per-CPU зон для универсального аллокатора ядерной памяти *uma(9)* в FreeBSD

1.5.1 API *uma(9)*

UMA представляет собой зонный аллокатор. Зона есть набор объектов, а в простейшем случае просто регионов памяти, обладающих идентичным размером. Зона может содержать как ограниченное число объектов, так и расти по мере запросов на аллокацию. Зоны создаются с помощью *uma_zcreate()*, после чего из зоны можно запрашивать память с помощью *uma_zalloc()* и освобождать с помощью *uma_zfree()*. Размер объектов, которые предоставляет зона, задаётся как аргумент к *uma_zcreate()*. Кроме размера, можно задать конструктор и деструктор для объектов, а также прочие свойства зоны, определяющие её внутренние параметры, так и внешнее поведение.

Первая цель данной работы состоит в том, чтобы предоставить возможность создавать зоны, которые будут возвращать не единичный регион памяти, а набор per-CPU регионов. Такое свойство будет задаваться флагом к *uma_zcreate()*.

1.5.2 Внутреннее устройство *uma(9)*

Очевидно, что аллокатор возвращающий регионы памяти одинакового размера представляет собой slab-аллокатор. То есть такой аллокатор, который нарезает какую-то последовательную область памяти на фрагменты установленного размера - плитки (англ. slab) и ведёт учёт того, какие элементы из этого наборы аллоцированы, а какие свободны. Регионы памяти под нарезку аллокатор *uma(9)* запрашивает у VM по-странично. Далее мы будем называть один такой регион слабом. Учёт того какие элементы в слабе свободны, а какие нет, ведётся во вспомогательной структуре *uma_slab*. Как правило, структура эта располагается непосредственно в слабе, который она описывает. Структура *uma_slab* хранит в себе указатель на начало

слаба, число свободных элементов и битсет, который описывает аллоцированные и свободные элементы.

Все слабы зоны разделяют такие общие характеристики как реальный размер одного элемента с учётом выравнивания (uk_rsize, читается real size), число элементов в одном слабе (uk_ipers, items per slab), сдвиг до структуры *uma_slab* относительно начала слаба (uk_pgoff, page offset). Так как эти характеристики общие, то хранятся они не в *uma_slab*, а в структурах описывающих зону.

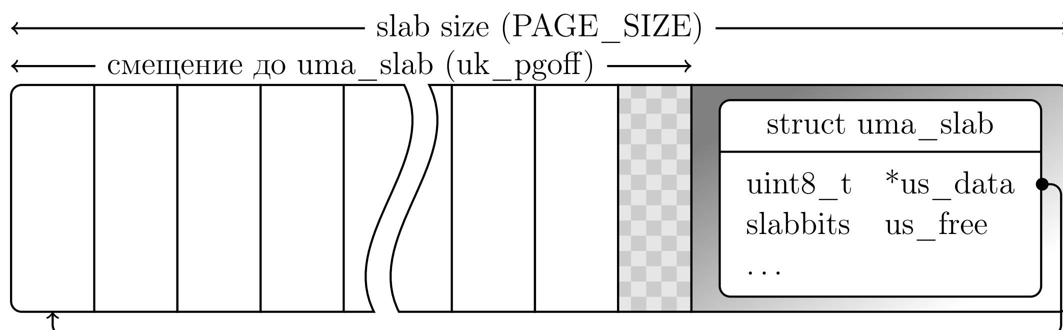

Рис. 1.1: Типичный layout отдельного слаба

В некоторых случаях размер элемента таков, что между последним элементом в слабе и структурой *uma_slab* остаётся неиспользуемая память. Если теряется достаточно большой объём памяти, то выгоднее аллоцировать структуру *uma_slab* из отдельной зоны и отдать всю страницу под слаб. Такой слаб мы далее будем называть слабом с внешним *uma_slab* или offpage слаб.

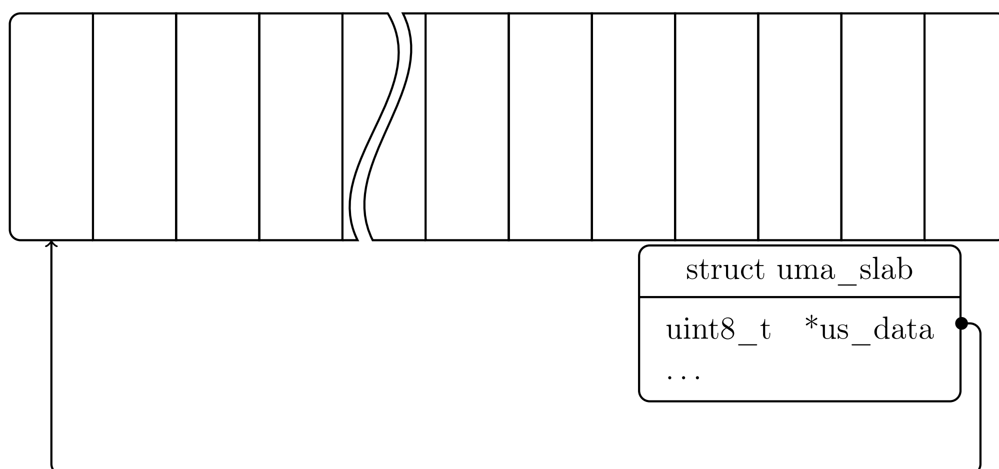

Рис. 1.2: offpage slab

Всякая *uma(9)* зона может содержать несколько слабов. Слабы собираются в три связанных списка: полностью заполненные аллокациями слабы, частично запол-

ненные и полностью свободные. Последние могут быть возвращены в VM в случае нехватки памяти в системе. Заголовки этих списков хранятся в структуре *uma_keg*, которую мы далее будем именовать кегом. В кеге также хранятся свойства общие для всех его слабов: размер аллокации, смещение до *uma_slab*, число аллокаций на один слаб, методы конструктор и деструктор и прочее. Внешним интерфейсом *uma(9)* является *uma_zone*. В простейшем случае зона содержит ровно один кег и кег принадлежит ровно одной зоне. В рамках данной работы рассмотрение более сложных случаев не потребуется, поэтому мы будем рассматривать структуры *uma_zone* и *uma_keg* как единое целое.

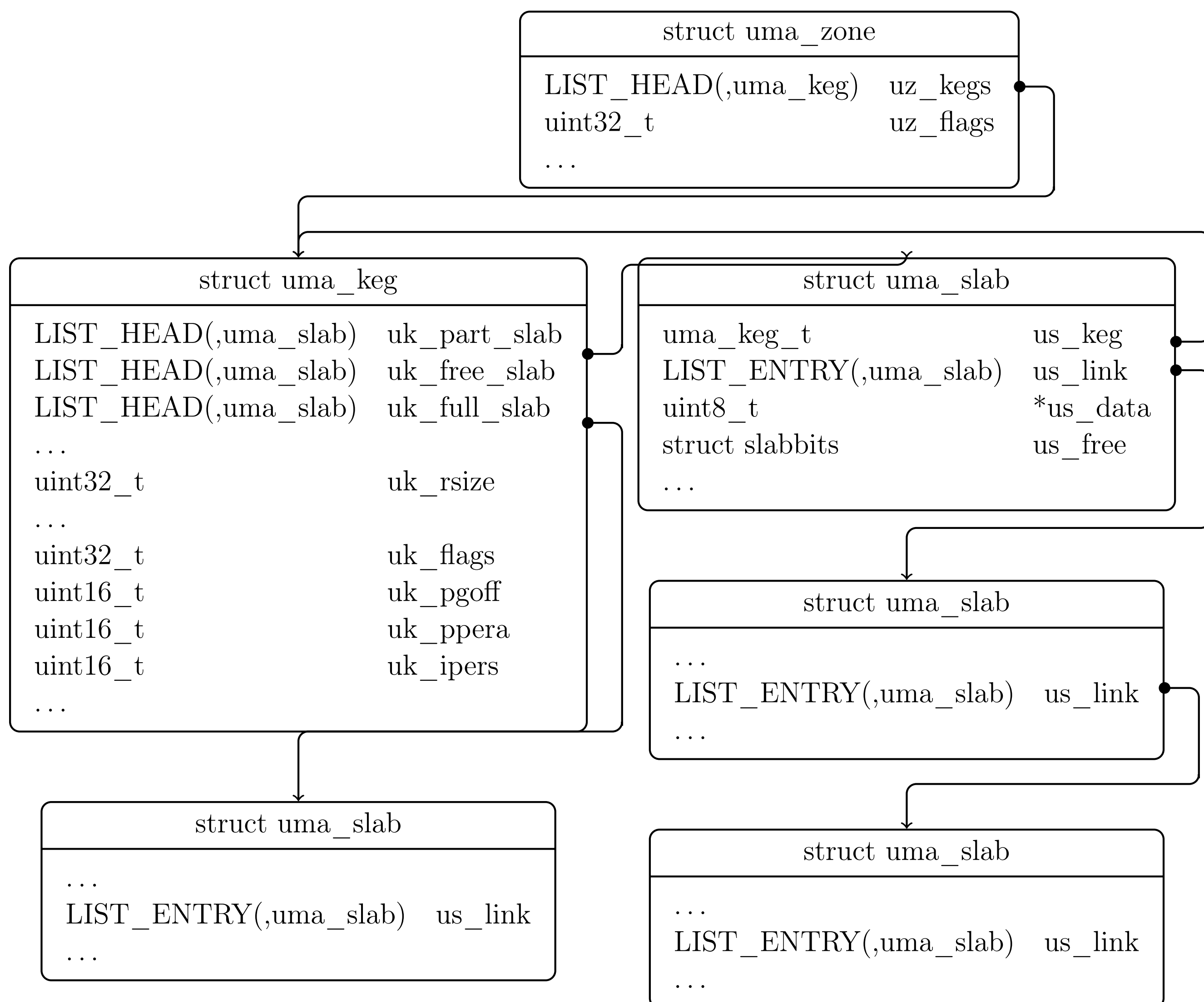

Рис. 1.3: Внутреннее устройство отдельной UMA зоны

1.5.3 Реализация аллокатора per-CPU памяти в рамках аллокатора *uma(9)*

Повторим постановку задачи более подробно, чем во введении. Требуется реализовать возможность указывать на стадии создания зоны, что создаваемая зона будет работать с per-CPU памятью, то есть нужен дополнительный флаг к *uma_zcreate()*. Назовём его UMA_ZONE_PCPU. Далее, значения, которые будет возвращать *uma_zalloc()*, из такой зоны должны быть не простыми указателями, а некими значениями, которые можно с помощью простой арифметики преобразовать в указатель на память приватную для текущего процессора.

Для решения поставленной задачи при создании зоны мы будем запрашивать у VM память не под один слаб, а под столько, сколько в системе процессоров. Далее мы будем называть такие слабы *per-CPU slab* или *процессорный слаб*. Эти слабы будут располагаться в виртуальной памяти в виде последовательного массива, и соответствующие элементы массива будут принадлежать соответствующим процессорам. Чтобы избежать cache sharing, размер одного процессорного слаба будет выровнен по величине процессорной кэш линии. Аллокатор будет работать над первым(нулевым) процессорным слабом так же как над обычным слабом, но всякая аллокация из первого процессорного слаба будет автоматически означать и аллокацию по аналогичному смещению и из всех прочих процессорных слабов. *uma_zalloc()* будет возвращать адрес из нулевого слаба, который в дальнейшем с помощью адресной арифметики может быть преобразован в адрес для текущего процессора.

Для реализации выше задуманного нам придётся либо нарезать одну страницу на процессорные слабы, либо запрашивать у VM более одной страницы памяти. Так как в современных компьютерах могут быть десятки процессоров, и есть тенденция к увеличению этого числа, то очевидно, что нарезая страницу на слабы, мы можем дойти до столь маленького размера слаба, что мы просто не сможем его нарезать на аллокации приемлемого размера, или он вообще станет меньше кэш линии, что опять же неприемлемо условиям поставленной задачи. Значит понадобится реализовать в UMA возможность запрашивать несколько страниц у VM. Однако, как мы увидим в главе 2, для реализации определённых оптимизаций нам потребуется жёстко задавать и размер процессорного слаба. А значит понадобится реализовать в UMA работу со слабами произвольного размера.

Так как аллокатор будет вести учёт аллокаций только в нулевом процессорном слабе, то нам потребуется только одна структура *uma_slab* на весь массив процессорных слабов, а значит наши новые per-CPU зоны будут всегда использовать offpage

слабы.

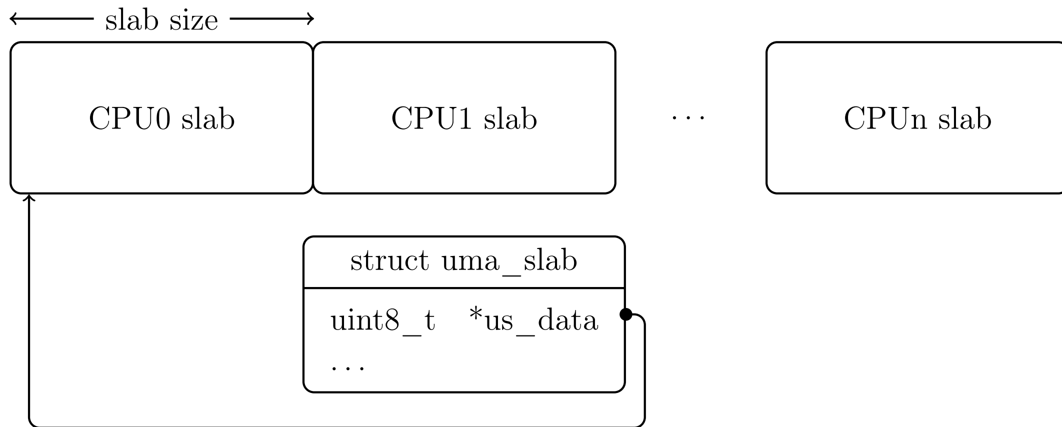

Рис. 1.4: Множественные слабы в VM аллокации для UMA_ZONE_PCPU зоны

Таким образом, в *uma_keg* мы вводим новые поля *uk_slabsize* и *uk_ppera*. Поле *uk_slabsize* задаёт размер слабов для данной зоны, который теперь произволен. Поле *uk_ppera* читается как pages per allocation и задаёт количество страниц запрашиваемое у VM под слабы, которое теперь может быть более одной [4].

```
type_t *base, *pcpu;

zone = uma_zcreate("zone", sizeof(var),
    NULL, NULL, NULL, NULL, 0, UMA_ZONE_PCPU);
...
base = uma_zalloc(zone, M_WAITOK);
...
critical_enter();
pcpu = (type_t *)((char *)(base) + sizeof(struct pcpu) * curcpu);
/*
 * Here *pcpu can be read or changed.
 */
critical_exit();
```

Рис. 1.5: Типичный случай использования новой функциональности

Флаг UMA_ZONE_PCPU при создании зоны означает, что для данной зоны UMA должен, во-первых, задать размер слаба равным *sizeof(struct pcpu)*, то есть

равным размеру структуры статического PCPU (см. главу 1.2.1), во-вторых, запрашивать у VM столько страниц, сколько необходимо для того, чтобы уместить *sizeof(struct pcpu) * mp_ncpus* байт, где *mp_ncpus* - переменная ядра хранящая число процессоров в системе. Почему размер процессорного слаба должен быть равным именно размеру *struct pcpu*, будет рассмотрено в главе 2.

Таким образом, первая из поставленных задач - реализация API для аллокации per-CPU памяти, решена. Подробнее см. [4].

Глава 2

per-CPU счётчики

2.1 Счётчики в BSD в ретроспективе

Во время исполнения операционной системы возникает необходимость собирать статистические данные о её работе, такие как количество и объём I/O операций на дисковых устройствах, количество пакетов и байт, переданных и полученных через сетевые устройства, и т.п.

В классической реализации UNIX, а позже BSD эта задача решалась элементарно, просто введением переменной, к которой прибавлялись данные с помощью оператора инкремента или присваивания.

```
ifp->if_opackets++;
ifp->if_obytes += m->m_pkthdr.len;
```

Рис. 2.1: Ведение статистики пакетов/байт в драйвере loopback в 4.4BSD Lite (net/if_loop.c)

В конце 1990-х - начале 2000-х гг ядра операционных систем стали становиться многотредовыми, то есть код ядра начал выполняться параллельно на нескольких процессорах. В таких условиях неатомарная операция обновления счётчика перестаёт быть безопасной. Однако, в первое время гранулярность локинга в ядрах оставляла желать лучшего, отдельные драйвера и даже подсистемы выполнялись под большими локами, поэтому, как правило, обновления счётчиков производилось под тем или иным локом и были безопасны. Позже, по мере того, как уровень параллелизма в ядрах рос, обновление некоторых счётчиков вышло из-под локов и перестало быть безопасным. В зависимости от важности счётчика разработчики либо просто

игнорировали проблему, либо заменяли обычное присваивание на атомарную операцию.

Атомарная операция подразумевает что процессор получает эксклюзивный доступ к шине памяти на время выполнения операции, а также то, что после окончания операции, в кэшах прочих процессоров те кэш-линии, которые соответствуют модифицированному адресу в памяти, будут инвалидированы. Очевидно, что выполнение таких операций пессимизирует производительность SMP системы, так как доступ процессоров к памяти сериализуется. В связи с этим сложилась тенденция использовать атомарные операции только для тех счётчиков, которые являются критически важными и неточность накопления данных в которых повлечёт неправильную работу операционной системы. Те же счётчики, которые собирают чисто статистическую информацию, временно оставили обычными операциями присваивания.

В конце 2000-х гигабитные сетевые интерфейсы стали повсеместны, а 10-ти гигабитные доступны. Некоторые статистические события, такие как приём и отправка пакета, стали происходить с частотой порядка миллиона раз в секунду. Выяснилось, что при такой частоте обновлений, во-первых, неатомарные счётчики теряют данные уже в объёмах заметных на практике, и во-вторых, что не атомарные счётчики тоже пессимизируют работу SMP системы. Оказалось, что устранение не атомарного инкремента может заметно увеличить производительность [5]. Это объясняется тем, что на архитектуре amd64 любая запись в память вызывает инвалидацию соответствующей кэш-линии в кэшах остальных процессоров. Если иной процессор в ближайшее время обратится к этой же памяти, то он столкнётся с непопаданием в кэш (cache miss). Если все процессоры работают над одной и той же задачей и постоянно инкрементируют одну и ту же ячейку в памяти, то они систематически сталкиваются с непопаданием в кэш. А на современных компьютерах доступ в память на пару порядков медленнее доступа к кэшу, отсюда и заметная разница в производительности вызванная такой безобидной операцией как ++, которая казалось бы даёт всего лишь пару ассемблерных инструкций.

2.2 Использование per-CPU памяти для реализации счётчиков

Становится очевидно, что от классического подхода, когда статистика ведётся всеми процессорами в единой ячейке памяти, надо уходить. Избежать постоянных непопаданий в кэш станет возможным, если каждый отдельный процессор будет

работать в своей области памяти, выровненной по длине кэш линии.

Одним из первых шагов в направлении ведения per-CPU статистики в FreeBSD становится перенос структуры *struct vmmeter*, в которой ведётся учёт системных событий, таких как прерывания, переключения контекста, системные вызовы и пр., из общей памяти в статический pcpu (см. главу 1.2.1). Очевидно, что нет возможности располагать все мыслимые счётчики в статическом pcpu. Число счётчиков невозможно предсказать на этапе компиляции или загрузки ядра. Новые счётчики могут аллоцироваться и разрушаться на этапе исполнения. Поэтому, необходим более общий метод, который позволил бы аллоцировать отдельные счётчики.

2.2.1 per-CPU счётчики в GNU Linux

Перед тем как приступить к реализации своего API, рассмотрим как решена аналогичная задача в других проектах. В ядре Linux для ведения системной статистики предоставлено API *percpu_counter* [6]. Счётчики инициализируются функцией *percpu_counter_init()*, разрушаются *percpu_counter_destroy()*. Инициализированный счётчик может быть обновлён с помощью функции *percpu_counter_add()* и его текущее значение возвращает функция *percpu_counter_sum()* в виде 64-битного значения.

```
struct percpu_counter {
        raw_spinlock_t lock;
        s64 count;
        s32 __percpu *counters;
};
```

Рис. 2.2: Структура percpu_counter в ядре GNU Linux (include/linux/percpu_counter.h)

Счётчик представляет из себя в первую очередь указатель *counters* на вектор указателей на отдельные 32-битные счётчики, расположенные в per-CPU памяти. Значения из множества per-CPU счётчиков аккумулируются в общем поле *count*, которое представляет из себя 64-битное целое. Перенос данных из per-CPU счётчика в общее поле очевидно требует синхронизации, для чего в структуре счётчика предусмотрен спинлок. Перенос данных осуществляется со всех процессоров когда требуется прочитать суммарное значение счётчика. Также очевидно, что 32 бит недостаточно

для того, чтобы получить счётчик, который гарантированно не переполнится в ближайшем будущем, поэтому когда при очередном инкременте per-CPU счётчика он приближается к переполнению, то данные из него переносятся в *count*.

Инкремент такого счётчика в обязательном порядке требует критической секции. Даже если архитектура предоставляет процессорную инструкцию, которая совершает инкремент непосредственно в памяти, без использования процессорного регистра, нет возможности выполнить обновление в одну инструкцию, т.к. перед обновлением необходимо, во-первых вычислить адрес per-CPU счётчика для текущего процессора, во-вторых провести операцию сравнения чтобы предусмотреть возможность переполнения.

2.2.2 API для системных счётчиков FreeBSD

Авторы поставили перед собой задачу разработать API счётчиков, которое имело бы минимальную привязку к реализации, что позволило бы в будущем пересмотреть реализацию в пользу иных технических решений, с сохранением API. Так, например, можно допустить, что дальнейшие исследования покажут однозначные преимущества аллокатора *vmem(9)* перед *uma(9)*. Может оказаться, что появятся машины с таким большим числом процессоров и сравнительно небольшим количеством оперативной памяти, что станет актуально отводить только 32 бита для каждого CPU, чтобы сэкономить память, аналогично тому как это реализовано в Linux. Возможно такая реализация потребуется только для машин с 32-битным машинным словом. Машины с неоднородным доступом к памяти (non-uniform memory access, NUMA) однозначно потребуют специального аллокатора per-CPU памяти. Таким образом, задекларируем API максимально скрывающее реализацию: счётчик представляет собой непрозрачный (opaque) тип, его можно аллоцировать, освободить, инкрементировать, обнулить и конечно получить его текущее значение.

```
counter_u64_t counter_u64_alloc(int wait);
void counter_u64_free(counter_u64_t c);
void counter_u64_add(counter_u64_t c, int64_t v);
void counter_u64_zero(counter_u64_t c);
uint64_t counter_u64_fetch(counter_u64_t c);
```

Рис. 2.3: Предлагаемое API *counter(9)* в FreeBSD

2.2.3 Реализация per-CPU счётчиков в FreeBSD с использованием критических секций

В реализации представленной в данной работе, счётчик *counter_u64_t* представляет собой указатель на *uint64_t*, который был аллоцирован из специальной per-CPU UMA зоны, реализация которой была представлена в главе 1.5.3. Эта зона, названная *uint64_pcpu_zone*, создаётся на раннем этапе загрузки ядра. Размер аллокации в ней равен sizeof(uint64_t), и для каждого процессора аллоцирован его собственный *uint64_t*. То есть, функция *counter_u64_alloc()* представляет собой обёртку к *uma_zalloc(uint64_pcpu_zone, ...)*.

Для того, чтобы инкрементировать или декрементировать счётчик, требуется вычислить смещение к ячейке памяти, принадлежащей текущему процессору, относительно базового адреса, а затем обновить эту ячейку. Вычисление смещения это операция умножения id текущего процессора на размер процессорного слаба. Последующее сложение смещения и базового адреса даёт адрес ячейки которую требуется обновить. Уже после выполнения операции умножения переход треда на другой процессор недопустим, так как величина смещения на другом процессоре будет отличной. Поэтому всю последовательность инструкций следует выполнять, запретив выполнение треда на других процессорах. Второе ограничение заключается в том, что на большинстве машин невозможно прибавить значение к ячейке памяти в одну инструкцию, требуется сначала загрузить значение в процессорный регистр, затем произвести сложение и затем запись в память. Это повышает требования к непрерываемости последовательности инструкций. Запрет на выполнение треда на других процессорах - недостаточное условие, необходим запрет выполнения других тредов на данном процессоре, то есть критическая секция. Таким образом, функция *counter_u64_add()* становится очень похожей на код представленный на рис. 1.5.

Ядерный интерфейс *critical(9)* в FreeBSD, который используется для защиты счётчиков, предоставляет "мягкие" критические секции (см. 1.1). Реализация представляет собой увеличение поля *critnest* в текущем треде, а указатель на текущий тред хранится в статическом pcpu. Доступ к текущему треду не требует синхронизации, т.к. данные приватны для текущего процессора. Такой вход в критическую секцию заметно дешевле полного запрета прерываний. В связи с вышесказанным, очевидным ограничением реализованной подсистемы *counter(9)* становится невозможность использования её в обработчиках прерываний. Это не столь серьёзное ограничение, так как в FreeBSD в контексте прерываний исполняется очень небольшое количество кода. Как правило, непосредственно обработчик только будит тред,

обслуживающий данное прерывание и возвращается.

2.2.4 Обновление счётчика без критической секции

Возможно ли прибавить значение к приватной ячейке памяти без входа в критическую секцию и при этом быть уверенным в том, что не произойдёт конфликта с другими тредами? Если машинное слово менее 64 бит, а счётчик 64-битный, то очевидно задача не решаема. Архитектура неспособная осуществлять инкремент непосредственно по адресу в памяти, также однозначно не позволяет решить задачу. Основная архитектура, для которой разрабатывается в настоящее время FreeBSD - amd64. Это машина с 64-битным словом и предоставляет инструкцию *add*, которая позволяет атомарно прибавить значение к ячейке памяти. Решаем ли задача в этих условиях?

Инструкция *add* обновляет память по адресу операнда, который может быть значением или регистром. В рассматриваемой задаче применим только регистр, так как адрес вычисляется динамически. Как уже было проиллюстрировано выше, адрес per-CPU аллокации в пямяти для каждого процессора свой, и без критической секции нет возможности вычислить адрес, поместить его в регистр, а потом обратиться к нему с помощью *add*.

Но есть два важных момента, которые позволяют нам решить задачу. Во-первых, архитектура amd64, как прямой наследник i386, предоставляет такую возможность как базированная адресация [7]. Адрес в памяти может быть адресован комбинацией двух регистров или регистра и значения. Процессор суммирует два значения для получения адреса. Во-вторых, когда FreeBSD выполняется на amd64 машине, то специальный регистр %gs всегда указывает на статический pcpu для текущего процессора. А как было отмечено в главе 1.2.1, все статические pcpu данные расположены в памяти в виде последовательного массива. Заметим, что процессорные слабы, описанные в главе 1.5.3, также расположены в памяти в виде массива.

Если жёстко задать размер процессорного слаба равным размеру статического pcpu, то есть sizeof(struct pcpu), то мы получим третье важное свойство: расстояние в памяти между приватными аллокациями из per-CPU UMA зоны также будет равным sizeof(struct pcpu). А значит, для любой аллокации из per-CPU UMA зоны, для любого процессора, смещение между статическим pcpu данного процессора и приватной аллокацией данного процессора будет таким же как и для прочих процессоров. Это означает, что если адресовать приватную аллокацию как (%gs + смещение), то величина смещения будет константная, независимо от того, на каком процессоре она

была вычислена.

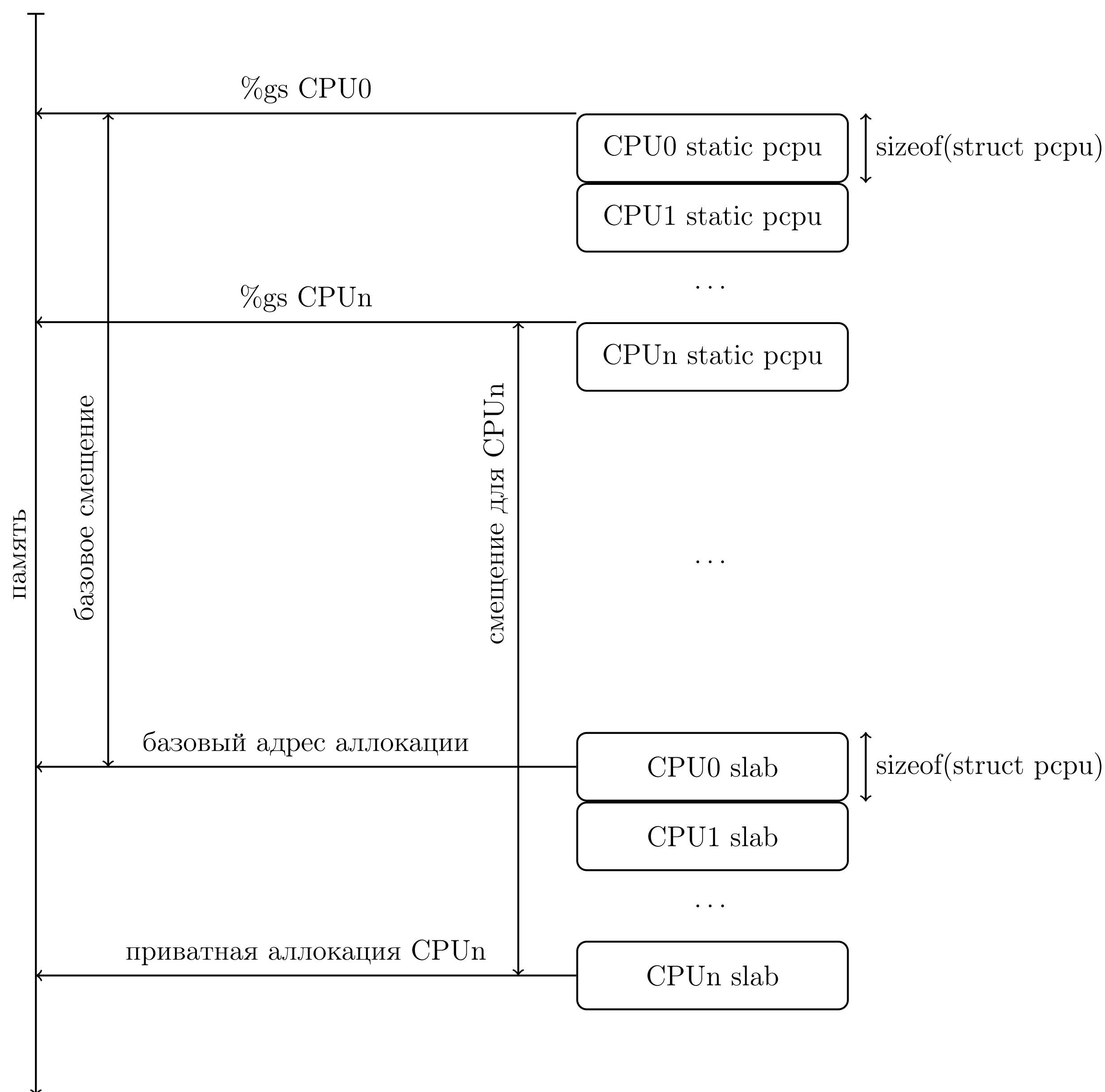

Рис. 2.4: Адресация аллокации в per-CPU UMA зоне относительно статического pcpu

Для вычисления константного смещения из базового адреса счётчика вычитается адрес нулевого элемента массива статических pcpu. Если тред будет отложен после вычисления смещения и продолжит исполнение на другом процессоре, то всё равно (%gs + смещение) будут указывать на правильную ячейку памяти. Таким образом, выполнение всех вышеперечисленных действий может быть произведено без критической секции.

```
static inline void
counter_u64_add(counter_u64_t c, int64_t inc)
{

        __asm __volatile("addq\t%1,%%gs:(%0)"
            :
            : "r" ((char *)c - (char *)&__pcpu[0]), "ri" (inc)
            : "memory", "cc");
}
```

Рис. 2.5: Реализация *counter_u64_add()* для amd64 (amd64/include/counter.h)

Таким образом вторая задача данной работы - реализация per-CPU счётчиков завершена [8].

2.3 Анализ результатов

2.3.1 Сравнение нового интерфейса, обычного инкремента и атомарного инкремента в синтетическом микробенчмарке

Для оценки результатов работы был написан простой синтетический тест. Загружаемый ядерный модуль создаёт тестовый счётчик с начальным значением равным нулю. Счётчик можем быть реализован тремя различными способами: *uint64_t* значение в памяти, обновляемое операцией '+=', *uint64_t* значение в памяти, обновляемое с помощью *atomic(9)*, *counter_u64_t*. Текущее значение счётчика доступно из юзерленд через *sysctl(9)*. Модуль реализует системный вызов принимающий два параметра. Первый параметр задаёт значение, которое следует прибавить к тестовому счётчику, а второй параметр сколько раз следует повторить эту операцию в цикле. Тестовая программа с помощью *fork(2)* порождает столько процессов потомков, сколько на машине есть процессоров. Каждый потомок привязывается к отдельному процессору, вызывает определённое число раз вышеописанный системный вызов и завершается. Родитель ожидает завершения всех детей с помощью *wait(2)*. В тесте мы измеряем реальное время выполнения родителя, а значит факт завершения всех потомков, и конечное значение счётчика.

Тестирование проводилось на машине с процессором Intel®Xeon®Processor E5620. Топологически это 4 ядерный процессор с 2-мя SMT тредами на ядро. Во всех тестах порождалось 4 процесса и они привязывались на чётные процессоры, чтобы избежать одновременной работы двух процессов на одном ядре. В первом тесте на один системный вызов приходилось 10 обновлений счётчика и очевидно, что основная работа, которую совершала машина, был вход в контекст ядра и возврат в юзерленд. Непосредственно работа со счётчиком составляла лишь малую часть процессорной активности. Однако даже в таком тесте видны преимущества новых счётчиков. Они не теряют данные и быстрее обычного инкремента на 20%. В сравнении с атомарным обновлением новые счётчики быстрее на 100%, то есть в два раза.

счётчик	значение счётчика	потери	время
+=	348314008	12.9%	4.24 с
atomic(9)	$4*10^8$	0	7.02 с
counter(9)	$4*10^8$	0	3.52 с

Рис. 2.6: Результаты микробенчмарка: 4 процесса, 10^7 системных вызовов, раундов цикла в вызове 10.

Второй тест представляет собой чистый микробенчмарк. Делается один системный вызов, в котором в плотном цикле совершается миллиард (10^9) обновлений счётчика. Такой тест даёт нагрузку чисто на счётчик, не тратя процессорные такты ни на что больше. Конечно, этот тест совершенно не отражает реальное использование статистических счётчиков при нормальной работе операционной системы, однако именно микробенчмарк иллюстрирует реальную разницу между альтернативными подходами к задаче. Результат для обычного инкремента показывает, что в плотном цикле пишущем в память практически только один процессор из четырёх успевает обновлять память, обновления от остальных трёх теряются. Также обычный инкремент оказывается на 46% медленнее нового счётчика, что объясняется постоянными не попаданиями в кэш. Атомарный инкремент, конечно, не теряет обновлений, однако из-за безумно высокой конкуренции за шину памяти оказывается в 20 с лишним раз медленнее нового счётчика.

счётчик	значение счётчика	потери	время
+=	1000662364	74.9%	3.65 с
atomic(9)	$4*10^9$	0	58.74 с
counter(9)	$4*10^9$	0	2.5 с

Рис. 2.7: Результаты микробенчмарка: 4 процесса, 1 системный вызов, раундов цикла в вызове 10^9.

2.3.2 Сравнение нового интерфейса, обычного инкремента и атомарного инкремента с помощью инструментов аппаратного анализа производительности

Современные процессоры предоставляют разработчику интерфейс для подробного анализа того, как именно выполняется код на процессоре. В FreeBSD доступ к этим данным осуществляется через драйвер *hwpmc(4)*, с помощью утилит *pmcstat(8)* и *pmccontrol(8)*.

Профилирование проводилось в условиях аналогичных описанным в главе 2.3.1: тот же процессор, тот же ядерный модуль, и аналогичные условия запуска, а именно 4 процесса, каждый привязан к одному ядру, каждый процесс делает один системный вызов, в котором осуществляется миллиард (10^9) обновлений.

В первую очередь нас интересует эффективность использования процессорных кэшей, ведь согласно нашим предположениям именно постоянные непопадания в кэш - основная проблема классических счётчиков.

	+=	*atomic(9)*	*counter(9)*
L1 retired loads hits	$1.65 * 10^9$	$1.69 * 10^9$	$2.81 * 10^{14}$
L1 prefetch requests	$6.73 * 10^7$	$1.04 * 10^9$	$6.98 * 10^4$
L1 prefetch misses	$4.84 * 10^7$	$1.03 * 10^9$	$2.58 * 10^4$

Рис. 2.8: Статистика использования L1 кэша

Наиболее близкий к ядру процессора кэш это L1 кэш, он же и самый небольшой по размеру. Чем больше утилизируется L1 кэш, тем эффективнее работает код. На используемом в тестах процессоре возможности по профилированию L1 кэша ограничены, тем не менее самое важное для нас событие, а именно попадания в кэш при чтении процессором памяти можно измерить. Как видно из первой строки таблицы 2.8 эффективность использования L1 кэша для счётчика на основе *counter(9)* на пять порядков выше эффективности для реализаций, где счётчик расположен в общей памяти. Также результаты профилирования иллюстрируют, что работа в общей памяти генерирует на три порядка больше префетчей, почти все из которых завершаются непопаданием в кэш.

	+=	*atomic(9)*	*counter(9)*
L2 requests (code & data)	$1.67 * 10^8$	$2.81 * 10^{14}$	$3.10 * 10^5$
L2 misses (code & data)	$1.33 * 10^8$	$2.81 * 10^{14}$	$1.83 * 10^5$
L2 requests loads	$6.21 * 10^7$	$1.03 * 10^9$	$8.60 * 10^4$
L2 requests load hits	$1.77 * 10^7$	$1.94 * 10^6$	$1.78 * 10^4$
L2 requests load misses	$4.83 * 10^7$	$1.03 * 10^9$	$5.97 * 10^4$
L2 requests RFOs	$3.65 * 10^7$	$1.04 * 10^9$	$3.74 * 10^4$
L2 requests RFO hits	$2.12 * 10^6$	$8.29 * 10^4$	$5.33 * 10^3$
L2 requests RFO misses	$3.56 * 10^7$	$1.04 * 10^9$	$2.71 * 10^4$
L2 requests prefetches	$6.62 * 10^7$	$2.10 * 10^9$	$1.35 * 10^5$
L2 requests prefetch hits	$1.56 * 10^7$	$1.04 * 10^9$	$3.27 * 10^4$
L2 requests prefetch misses	$4.87 * 10^7$	$1.06 * 10^9$	$1.01 * 10^5$
L2 requests (all)	$1.30 * 10^8$	$2.81 * 10^{14}$	$2.16 * 10^5$
L2 demand requests	$6.38 * 10^7$	$1.03 * 10^9$	$8.74 * 10^4$
L2 prefetch requests	$6.02 * 10^7$	$2.08 * 10^9$	$1.14 * 10^5$
L2 store RFO requests	$3.83 * 10^7$	$4.45 * 10^6$	$3.83 * 10^4$
L2 store RFO hits	$2.33 * 10^6$	$3.60 * 10^5$	$1.46 * 10^4$
L2 demand lock RFO requests	101	$1.04 * 10^9$	108
L2 demand lock RFO hits	66	$6.73 * 10^4$	81
L2 demand lock RFO miss	38	$1.03 * 10^9$	41

Рис. 2.9: Статистика использования L2 кэша

При беглом осмотре таблицы 2.9 бросается в глаза, что все значения в столбце *counter(9)*, как правило, на несколько порядков меньше, чем соответствующие значения в первых двух столбцах. Эта особенность только подтверждает вывод сделанный из таблицы 2.8, что в данном тесте *counter(9)* столь эффективно утилизирует L1 кэш, что практически не пользуется L2 кэшом.

Внимательное сравнение значений попаданий и непопаданий в кэш для атомарного инкремента показывает, что процент попаданий вовсе стремится к нулю. Фактически атомарный инкремент в общей памяти не использует кэш.

Для неатомарного инкремента соотношения попаданий и непопаданий лучше, но не стоит забывать, что за каждым попаданием скрывается потеря обновления сделанного другим процессором. Действительно, как было отмечено в таблице 2.7, в плотном цикле лишь одна четверть (в случае четырёх процессоров) обновлений достигает памяти. Аналогичное приблизительное соотношение в одну четверть наблюдается и в сравнении количества попаданий при загрузке из L2 кэша (L2 requests load hits) к общему количеству загрузок из L2 кэша (L2 requests loads).

2.3.3 Использование нового интерфейса в реальных задачах

Одной из предпосылок к началу данной работы были результаты из [5], где было отмечено, что устранение ведения статистики из TCP/IP стека ведёт к значительной экономии процессорного времени, что в определённых условиях приводит и к увеличению скорости форвардинга пакетов. Логично, что одним из первых пользователей нового API в ядре FreeBSD стал TCP/IP стек [9]. Структуры *struct ipstat* и *struct tcpstat* описанные ещё в [10], перестают использоваться в ядре как таковые, и остаются только для обеспечения интерфейса между ядром и юзерленд для таких утилит как *netstat(1)*. Реальная же статистика теперь ведётся в массивах из *counter_u64_t*, размер которых соответствует числу полей в *struct ipstat* и *struct tcpstat*.

Для оценки производительности новых счётчиков был совершён следующий тест. Генератор трафика Ixia посылал на тестируемую машину трафик, в объёме заведомо превышающем её возможности по обработке. Тестируемая машина просто принимала пакеты и обрабатывала их вплоть до протокольного уровня, без дальнейшей передачи их в сокеты или форвардинга. В тесте измерялось максимальное количество пакетов в секунду, которое машина способна обработать и загрузка процессора при этом.

	тыс. пакетов/с	CPU idle
+=	2424	6%
counter(9)	2428	55%

Рис. 2.10: Счётчики в TCP/IP стеке

Хотя статистически значимой прибавки в производительности системы и не произошло, но освободилось значительное количество процессорного времени. Вероятно, какие-то иные лимитирующие факторы в сетевом стеке или в драйвере сетевой карты препятствуют увеличение пропускной способности, несмотря на появление дополнительных процессорных циклов.

2.4 Дальнейшая работа

2.4.1 Применение *counter(9)* в ядре FreeBSD

Авторы видят дальнейшее применение нового API в первую очередь в сетевом стеке, где частота событий в секунду традиционно на порядки превосходит частоту

событий в дисковом I/O. В ближайшее время новые счётчики будут применены в пакетных фильтрах *ipfw(4)*, *pf(4)*, в модульном сетевом стеке *netgraph(4)*. Счётчики пакетов и байт в *struct ifnet* отражающей сетевой интерфейс также остро нуждаются в переходе на *counter(9)*, однако этот переход затруднён тем, что изменение в API и ABI затронет не только ядро, но и юзерленд утилиты, в том числе и те, что не входят в дистрибуцию FreeBSD.

Структура *struct vmmeter*, которая сейчас встроена в статический pcpu, также может быть конвертирована на *counter(9)*. Это будет полезно как для унификации статистики в ядре, так и для того, чтобы существенно уменьшить размеры статического pcpu. Работа в этом направлении уже ведётся, см. [11].

2.4.2 Лёгкий счётчик ссылок на основе *counter(9)*

Как иллюстрирует данная работа, переход от средств синхронизации основанных на атомарных инструкциях к приватным процессорным данным позволяет значительно увеличить производительность в некоторых задачах. В данной работе мы решили такую частную задачу как статистические счётчики. Другой, весьма актуальной задачей является множественный параллельный доступ к данным на чтение, при условии что доступ на запись случается крайне редко. Решение этой задачи "в лоб", то есть использование read/write lock, сталкивается с большими издержками на синхронизацию. На первый взгляд множественные читающие треды не конкурируют непосредственно за лок, т.к. в отсутствие пишущего треда, читатели всегда получают лок беспрепятственно. Однако это беспрепятственное взятие лока означает непопадание в кэш с вероятностью 100%, то есть множественные читатели агрессивно конкурируют за линию кэша. Более сложные решения этой задачи заключаются в том, что пишущий тред косвенным образом отслеживает то, что читатели перестали держать ссылки на данные, которые следует удалить или изменить. Примером такого решения является RCU [12], патентованная IBM технология.

Если удастся реализовать счётчик ссылок на данные через per-CPU счётчики, то такое решение будет лишено издержек на синхронизацию и не должно нарушать патент на RCU.

Литература

[1] Реализация per-CPU аллокатора в NetBSD
http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/kern/subr_percpu.c

[2] Аллокатор общего назначения *vmem(9)*
http://svnweb.freebsd.org/base/head/sys/kern/subr_vmem.c
http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/kern/subr_vmem.c

[3] LWN article: Driver porting: per-CPU variables
http://lwn.net/Articles/22911/

[4] Изменение r249264 в репозитории FreeBSD: реализация per-CPU зон в аллокаторе UMA
http://svnweb.freebsd.org/base?view=revision&revision=249264

[5] Профайлинг форвардинга через 10G интерфейсы Intel под FreeBSD
http://lists.freebsd.org/pipermail/freebsd-performance/2012-July/004613.html

[6] Реализация per-CPU счётчика в GNU Linux
http://lxr.linux.no/#linux+v3.10.7/include/linux/percpu_counter.h
http://lxr.linux.no/#linux+v3.10.7/lib/percpu_counter.c

[7] The 80x86 Addressing Modes
http://cs.smith.edu/ thiebaut/ArtOfAssembly/CH04/CH04-2.html

[8] Изменение r249268 в репозитории FreeBSD: новое API *counter(9)*
http://svnweb.freebsd.org/base?view=revision&revision=r249268

[9] Изменение r249276 в репозитории FreeBSD: *counter(9)* для ведения статистики в TCP/IP стеке
http://svnweb.freebsd.org/base?view=revision&revision=r249276

[10] Gary R. Wright, W. Richard Stevens,
TCP/IP Illustrated, Volume 2: The Implementation
ISBN-10: 0-201-63354-X, ISBN-13: 978-0-201-63354-2

[11] Reworking vmmeter
http://lists.freebsd.org/pipermail/freebsd-arch/2013-July/014471.html

[12] What is RCU, Fundamentally?
http://lwn.net/Articles/262464/