Для цитирования: Кожин А.С., Недбайло Ю.А. Методы оптимизации времени доступа в общий кэш многоядерного микропроцессора // Вопросы радиоэлектроники. 2017. № 3. С. 27–32.
УДК 004.318

А.С. Кожин[1], Ю.А. Недбайло[1,2]

[1] ЗАО «МЦСТ», [2] ПАО «ИНЭУМ им. И.С. Брука»

МЕТОДЫ ОПТИМИЗАЦИИ ВРЕМЕНИ ДОСТУПА В ОБЩИЙ КЭШ МНОГОЯДЕРНОГО МИКРОПРОЦЕССОРА

Рассмотрена проблема роста времени доступа при масштабировании распределенного общего кэша. Описаны существующие методы ее решения, такие как привязка страниц к банкам кэша, создание копий строк в ближайшем к ядру банке и миграция строк. Произведена оценка влияния реализации кэша на производительность 16-ядерного процессора.

Ключевые слова: *архитектура, многоядерность, подсистема памяти, общий кэш, NUCA.*

Введение

Кэши в микропроцессорах играют важную роль, снижая время доступа в память и уменьшая частоту обращений к ней. В настоящее время большинство многоядерных процессоров имеют, помимо небольших приватных кэшей, каждый из которых используется только одним ядром, также и большой общий кэш, объем которого доступен всем ядрам и разделен на банки, распределенные по кристаллу.

Изучение зависимости частоты промахов в кэше в разных задачах от его размера (рис. 1) позволяет сделать два вывода. Во-первых, чем больше доступный объем кэша, тем частота промахов меньше; нет абсолютного достаточного объема – любое увеличение дает заметный выигрыш в каких-то задачах. Во-вторых, разные задачи (а иногда и треды одной задачи) имеют совершенно разную потребность в объеме кэша, поэтому основной объем кэша лучше динамически распределять между тредами, чем наделять ядра только приватными кэшами фиксированного размера.

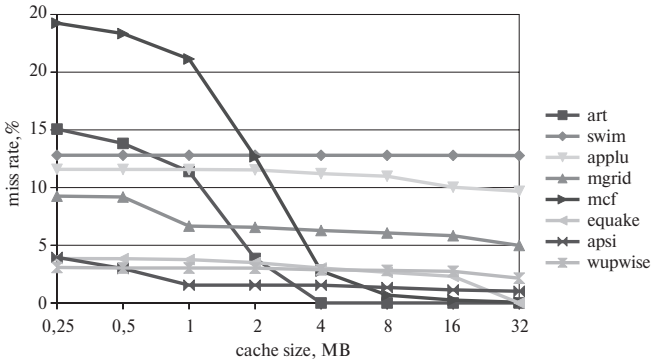


Рисунок 1. Частота промахов в кэше в зависимости от его объема в некоторых тестах пакета SPEC CPU2000 на симуляторе процессора «Эльбрус»

Таким образом, общий кэш является самым предпочтительным вариантом реализации кэша последнего уровня многоядерного процессора, давая наибольший доступный объем и максимальную гибкость в его распределении между тредами. Однако с ростом количества ядер обостряется ряд проблем его реализации.

В данной статье мы рассмотрим проблему увеличения времени доступа из-за увеличения среднего расстояния между ядрами и банками кэша и способы ее решения с помощью оптимизации выбора банков, в которых хранятся данные.

Распределенный общий кэш

В небольших процессорах – с двумя или четырьмя ядрами – общий кэш делается единым устройством, но начиная с некоторого числа ядер обычно возникает проблема нехватки его пропускной способности. Чтобы решить эту проблему, кэш делят на банки, распределенные по кристаллу (рис. 2). При условии, что сеть, соединяющая ядра и банки кэша, не станет узким местом, пропускная способность кэша будет достаточной для любого числа ядер, если число банков пропорционально увеличивать.

Однако и время доступа в распределенный кэш увеличивается пропорционально размеру системы, поскольку увеличивается среднее число шагов, совершаемых по сети. Проблема кроется в случайном распределении данных по банкам в традиционной схеме (называемой S-NUCA): положение каждой кэш-строки (так называемый home-банк) зависит только от физического адреса, который, в свою очередь, определяется случайной динамикой выделения памяти операционной системой (ОС)

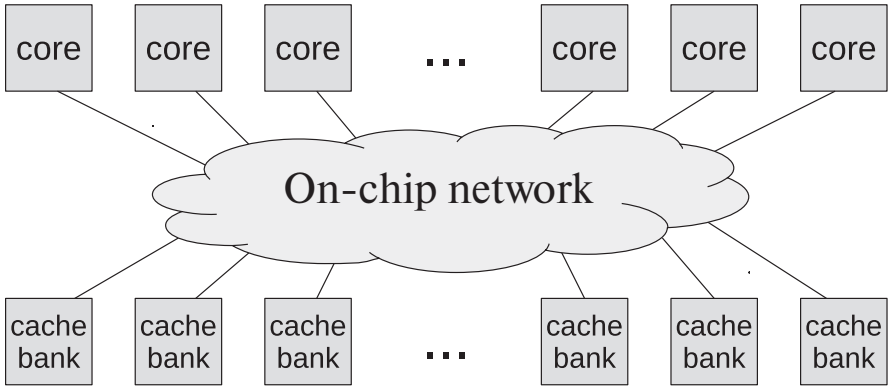


Рисунок 2. Концепция распределенного общего кэша

и не зависит от того, какое ядро пользуется этими данными.

Избавление тем или иным образом от этой случайности и расположение данных вблизи использующих их ядер является известным путем решения указанной проблемы. При этом существуют три основных подхода, которые будут рассмотрены далее:

1. Привязка страниц памяти к банкам кэша силами ОС.
2. Создание копии строки в ближайшем банке при обращении к ней.
3. Миграция строки в ближайший банк при обращении к ней.

Существующие методы

Привязка страниц памяти к банкам

Физический адрес запроса программы в память, как правило, формируется из виртуального путем обращения в таблицу страниц. В этой же таблице можно хранить различные свойства каждой страницы, в т.ч. номер банка в кэше, и передавать их вместе с запросом. Тогда номер банка не будет привязан к физическому адресу, и операционная система сможет задавать его произвольно при выделении памяти – например, выставлять банк, ближайший к тому ядру, на котором запущен процесс. Также можно ввести признак отсутствия привязки страницы к какому-либо банку, означающий традиционное рассеяние этой страницы по банкам, чтобы ей использовался весь кэш. Подобные оптимизации применяются в многопроцессорных системах с распределенной памятью (NUMA).

Более развитая версия этой оптимизации, ориентированная на «плиточную» топологию процессора (рис. 3а), описана в [1]. Здесь страницы могут иметь несколько градаций рассеяния по банкам – например, для 16 банков предлагается 1, 4 и 16. Для промежуточной градации и заданного (в таблице страниц) номера ядра номер банка вычисляется на основе нескольких битов адреса по принципу Rotational Interleaving (рис. 3б).

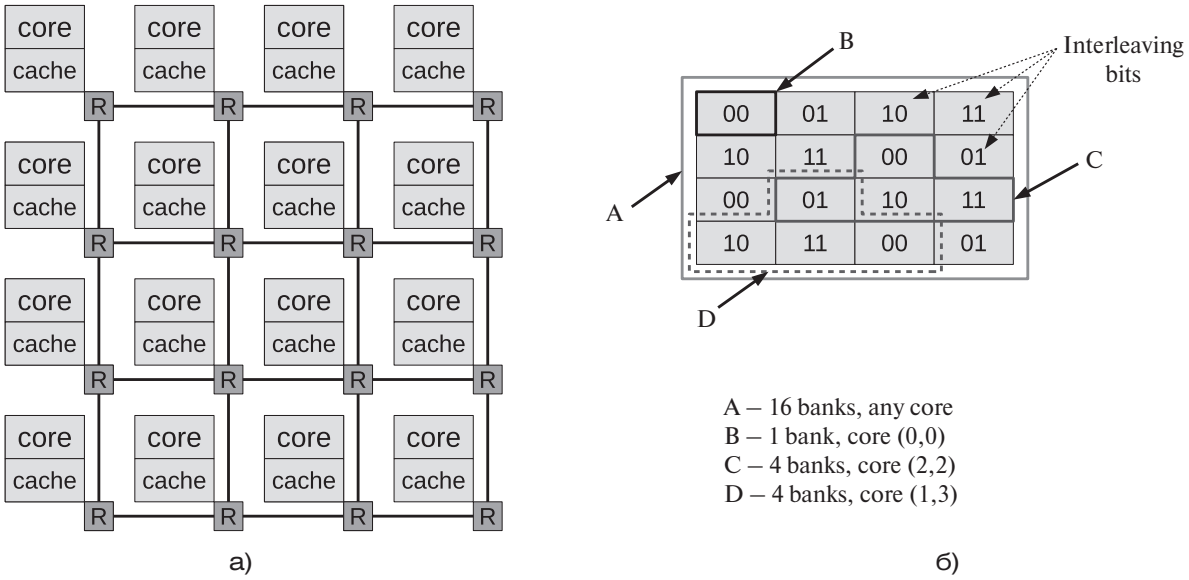


Рисунок 3. Reactive NUCA: а – «плиточная» топология; б – Rotational Interleaving

Такой подход имеет два недостатка. Во-первых, он требует модификации ОС и либо какой-то эвристики (определяющей, какие страницы нужно хранить вблизи ядра, а какие рассеивать по банкам), либо жертвования доступным программе объемом. Во-вторых, оптимизация перестанет хорошо работать при случайной пересадке программ на другие ядра, что часто бывает в многозадачной ОС, если не привязывать задачи к ядрам явно.

Создание копий строк в ближайшем банке

Альтернативой привязке данных к желаемым банкам кэша при выделении памяти является их перемещение между банками при обращении к ним. Такой подход решает проблему времени доступа, даже когда программы пересаживаются планировщиком ОС с одного ядра на другое. Но этому существует препятствие. Общий кэш больших процессоров чаще всего делается инклюзивным: каждая строка каждого приватного кэша также имеет копию в соответствующем ее адресу банке общего кэша. Эта копия может быть без актуальных данных, но обязательно с информацией о том, кто владелец актуальной версии и в каких вышестоящих по иерархии кэшах (или банках этого же кэша) могут быть еще копии. Это дает возможность поддерживать когерентность всех кэшей процессора без использования плохо масштабируемых широковещательных сообщений, но не позволяет просто перемещать строку между банками, как хотелось бы.

Тем не менее даже в инклюзивном кэше можно создавать копию используемой ядром строки в ближайшем к нему банке. Копирование каждой строки уменьшит емкость кэша как минимум вдвое, и такая оптимизация вряд ли целесообразна. Следовательно, требуется каким-то образом определять, к каким данным время доступа достаточно критично. Некий эвристический механизм предложен в [2], в среднем он дает выигрыш в несколько процентов и имеет соответствующую стоимость.

Миграция строк в ближайший банк

Общий кэш не обязательно делать инклюзивным. До какого-то числа ядер можно использовать широковещательные сообщения для поддержания когерентности, потом можно реализовать отдельный справочник, хранящий всю информацию о копиях. Это и само по себе повышает эффективность кэшей, снижая частоту промахов [3], и позволяет реализовать перемещение строк между банками общего кэша, не жертвуя его емкостью.

Поскольку, в отличие от рассмотренной выше схемы, при миграции строка присутствует в общем кэше в единственном экземпляре, краеугольным камнем становится вопрос ее дальнейшего перемещения между банками при вытеснении из самого ближайшего. Если ее просто вычеркивать, кэш превратится в набор приватных. Мы можем выделить три подхода к вытеснению в этом случае:

1. Victim-миграция – в home-банк.
2. Кооперативное кэширование – в случайный банк.
3. CloudCache – по цепочке ближайших банков.

Victim-миграция

Упомянутый когерентный справочник целесообразно делать таким же распределенным, как и общий кэш, и совмещать их, т.е. просмотр кэша и справочника производить одновременно и в одном месте – home-банке. Тогда и вытеснять строку из ближайшего к ядру банка лучше в home-банк, иначе она будет продолжать занимать место в справочнике и обращение к ней (после просмотра справочника) будет требовать дополнительного времени и сетевого трафика. Такая схема известна как victim-миграция [4]. Изначальный вариант предлагает набор правил, по которым строки перемещаются между банками и выбираются жертвы для вытеснения. Другая статья [5] предлагает еще пять вариантов.

В общем случае можно считать, что каждый банк динамически разделяется на приватную и общую части и общая часть выступает в роли victim-кэша для приватных (рис. 4). Главным вопросом остается пропорция, в которой их нужно делить, или по каким признакам отдавать приоритет при замещении. Для этого применимы те же механизмы, которые предлагаются для кооперативного кэширования.

Кооперативное кэширование

Рассмотренная выше схема основывается на распределенном общем кэше, как бы добавляя в него приватную часть. Это имеет свои плюсы, но, вероятно, хорошо работает только с приватными частями небольшого размера. Если некоторый банк будет полностью занят приватной частью, запросы в общий кэш соответствующего диапазона адресов будут всегда промахиваться. От этой проблемы избавлено так называемое кооперативное кэширование.

На основе приватных кэшей и отдельного от них механизма когерентности был предложен более гибкий вариант общего кэша. Поскольку home-банка как такового в этом случае нет, нет и большой разницы, в какой банк отправлять строку при вытеснении из ближайшего. В простейшей реализации он выбирается случайно, при определенном разбиении банков на приватную и общую части – арбитром, учитывающим соответствующие весовые коэффициенты (например, Weighted Round Robin).

Как и в предыдущем случае, возникает вопрос динамического определения оптимальных

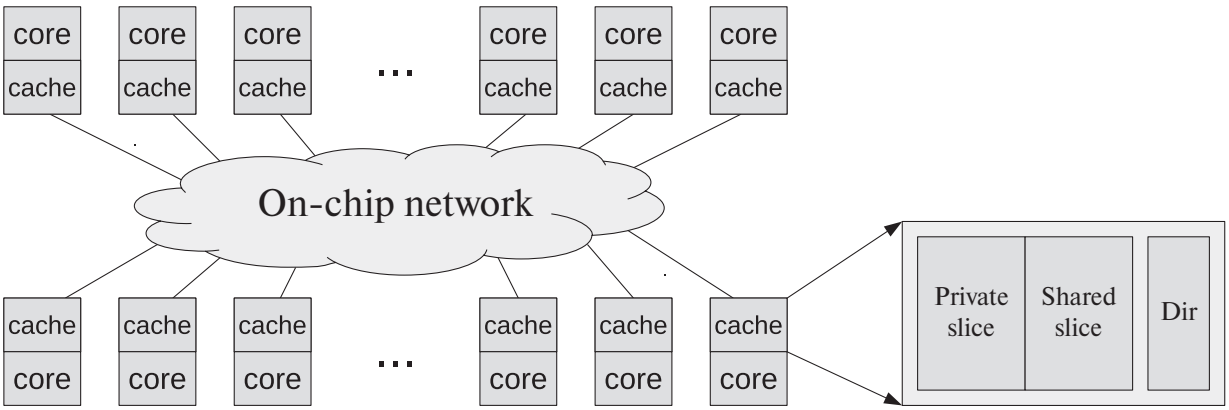


Рисунок 4. Деление общего кэша на приватную и общую части и справочник

коэффициентов. Простой вариант предлагается в [6] – использовать для разбиения way-partitioning, т.е. распределять колонки кэша между приватной и общей частями и при попадании в самую старую строку одной из частей инкрементировать или декрементировать некоторый счетчик, а по достижении счетчиком заданного порога менять коэффициенты разбиения в соответствующую сторону.

В целом кооперативное кэширование гибче, чем victim-миграция, но требует больше места в справочнике и имеет большее время доступа в общую часть. Поскольку принципиальных отличий между этими схемами нет, наверняка можно сконструировать и некоторую гибридную схему, сочетающую их качества.

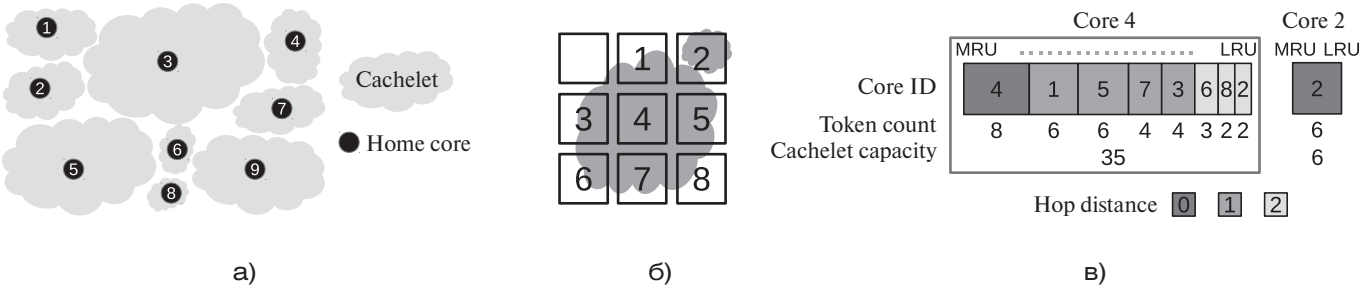
CloudCache

Victim-миграция и кооперативные кэши наделяют один ближайший к ядру банк свойствами приватного кэша, позволяя работать с ним непосредственно, а остальные банки используются как распределенный общий кэш с доступом посредством (или параллельно с просмотром) справочника.

В [7] предлагается принципиально другой подход – динамически привязывать к ядру части нескольких банков, позволяя работать с ними так же непосредственно – CloudCache (рис. 5). Все части одного ядра образуют порцию (cachelet), которая представляет собой список банков и размеров разделов, принадлежащих этому ядру. Для определения оптимального размера порции каждое ядро предлагается снабжать специальным устройством сбора статистики Utility Monitor [8]. Всех деталей реализации статья, предлагающая данный метод, не раскрывает, но в целом схема определенно дорогая и сложная, и вызывают вопросы ее масштабируемость и дополнительный сетевой трафик. Тем не менее сама идея использования нескольких банков вместо одного ближайшего выглядит реализуемой и заслуживает внимания.

Моделирование

Для сравнения и оценки преимуществ приватных, общего и динамически разделяемого (гибридного) кэшей на программном симуляторе 16-ядерного процессора «Эльбрус», аналогичного модели «Эльбрус-8С2», моделировался запуск одного из тестов пакета SPEC CPU2000 на одном ядре (рис. 6). Во всех конфигурациях суммарный объем L3-кэша составляет 32 МБ, но в приватной конфигурации ядру доступны только ближайшие 2 МБ. Таким образом, общий кэш предоставляет ядру в 16 раз больший объем, но имеет большее время доступа, поэтому в большинстве задач общий кэш показывает себя лучше, но в двух из восьми – приватные лидируют.



а) б) в)

Рисунок 5. CloudCache: а – распределение кэша в процессоре с девятью активными ядрами; б – пример распределения двух порций по банкам; в – представление порций в ядре

Простейший гибридный кэш, создающий копии в ближайшем банке, в каждом тесте лучше простого общего и в среднем превосходит приватные на 3,7% и общий на 2,3%. При большем количестве ядер разница должна быть более выраженной.

Заключение

Описанные в данной статье оптимизации времени доступа хорошо применимы к распределенному общему кэшу и предлагают разные комбинации производительности, стоимости и гибкости. Эти методы можно сочетать, чтобы добиться оптимального соотношения характеристик. Проблема роста времени доступа таким образом может быть значительно уменьшена.

В зависимости от параметров системы влияние времени доступа в общий кэш на производительность может варьироваться от единиц до десятков процентов, поэтому количество ядер, при котором такие оптимизации становятся целесообразными, в каждом конкретном семействе процессоров свое и может определяться экспериментально.

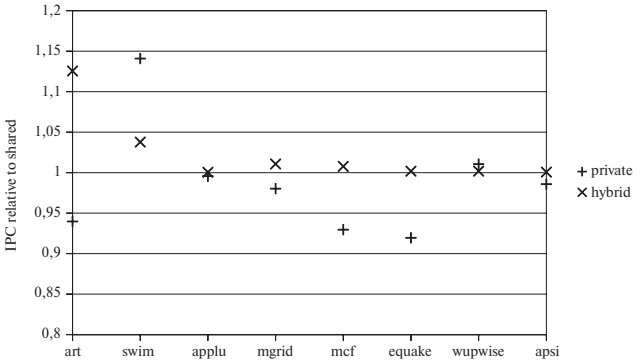


Рисунок 6. Разница производительности тестов SPEC CPU2000 на симуляторе процессора «Эльбрус» в зависимости от реализации кэша третьего уровня

СПИСОК ЛИТЕРАТУРЫ

1. Hardavellas N., Ferdman M., Falsafi B., Ailamaki A. Reactive NUCA: Near-optimal block placement and replication in distributed caches. SIGARCH Comput. Archit. News, 2009, no. 37 (3), pp. 184–195.
2. Beckmann B.M., Marty M.R., Wood D.F. ASR: Adaptive selective replication for CMP caches. Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '39, Washington, DC, USA, 2006, pp. 443–454.
3. Кожин А.С., Недбайло Ю.А. Оптимизация общего кэша третьего уровня микропроцессора «Эльбрус-8С» // Вопросы радиоэлектроники. 2015. № 3 (3). С. 21–30.
4. Asanovic K., Zhang M. Victim migration: Dynamically adapting between private and shared CMP caches. Technical report, Computer Science and Artificial Intelligence Laboratory, 2005.
5. Zhao L., Iyer R., Upton M., Newell D. Towards hybrid last level caches for chip-multiprocessors. SIGARCH Comput. Archit. News, 2008, no. 36 (2), pp. 56–63.
6. Herrero E., Gonzalez J., Canal R. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA'10, New York, 2010, pp. 419–428.
7. Lee H., Cho S., Childers B.R. CloudCache: Expanding and shrinking private caches. Proceedings of the 2011 IEEE17th International Symposium on High Performance Computer Architecture, HPCA'11, Washington, 2011, pp. 219–230.
8. Qureshi M.K., Patt Y.N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'39, Washington, 2006, pp. 423–432.

ИНФОРМАЦИЯ ОБ АВТОРАХ

Кожин Алексей Сергеевич, старший инженер, АО «МЦСТ», 119334, Москва, ул. Вавилова, д. 24, тел.: 8 (499) 135-31-08, e-mail: alexey.s.kozhin@mcst.ru.
Недбайло Юрий Александрович, старший инженер, АО «МЦСТ», ПАО «ИНЭУМ им. И.С. Брука», 119334, Москва, ул. Вавилова, д.24, тел.: 8 (916) 936-86-70, e-mail: nonsens@inbox.ru.

For citation: Kozhin A.S., Nedbailo Yu.A. Methods of shared cache access latency optimization in chip multiprocessors. Voprosy radioelektroniki, 2017, no. 3, pp. 27–32.

A.S. Kozhin, Yu.A. Nedbailo

METHODS OF SHARED CACHE ACCESS LATENCY OPTIMIZATION IN CHIP MULTIPROCESSORS

The paper is dedicated to shared cache scaling problems in a CMP with the focus on cache access latency. The main existing methods of its mitigation are discussed, including page colouring, victim replication, and victim migration. The effect of cache implementation methods on the performance of a 16-core processor is evaluated.

Keywords: architecture, many-core, memory subsystem, shared cache, NUCA.

REFERENCES

1. Hardavellas N., Ferdman M., Falsafi B., Ailamaki A. Reactive NUCA: Near-optimal block placement and replication in distributed caches. *SIGARCH Comput. Archit. News,* 2009, no. 37 (3), pp. 184–195.
2. Beckmann B.M., Marty M.R., Wood D.F. ASR: Adaptive selective replication for CMP caches. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '39, Washington,* DC, USA, 2006, pp. 443–454.
3. Kozhin A.S., Nedbailo Yu.A. Optimizing the inclusive shared L3 cache in «Elbrus-8S» microprocessor. *Voprosy radioelektroniki,* 2015, no. 3 (3), pp. 21–30.
4. Asanovic K., Zhang M. Victim migration: Dynamically adapting between private and shared CMP caches. *Technical report, Computer Science and Artificial Intelligence Laboratory,* 2005.
5. Zhao L., Iyer R., Upton M., Newell D. Towards hybrid last level caches for chip-multiprocessors. *SIGARCH Comput. Archit. News,* 2008, no. 36 (2), pp. 56–63.
6. Herrero E., Gonzalez J., Canal R. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA'10,* New York, 2010, pp. 419–428.
7. Lee H., Cho S., Childers B.R. CloudCache: Expanding and shrinking private caches. *Proceedings of the 2011 IEEE17th International Symposium on High Performance Computer Architecture, HPCA'11, Washington,* 2011, pp. 219–230.
8. Qureshi M.K., Patt Y.N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '39, Washington,* 2006, pp. 423–432.

AUTHORS

Kozhin Aleksey, senior engineer, JSC «MCST», Moscow, 119334, Russian Federation, 24, Vavilova st., (499) 135-31-08, e-mail: alexey.s.kozhin@mcst.ru.

Nedbailo Yuriy, senior engineer, JSC «MCST», PJSC «Brook INEUM», Moscow, 119334, Russian Federation, 24, Vavilova st., (916) 936-86-70, e-mail: nonsens@inbox.ru.