Для цитирования: Шмейлин Б.З. Методы замещения данных в кэш-памяти // Вопросы радиоэлектроники. 2017. № 2. С. 43–48.
УДК 004.318

**Б.З. Шмейлин**[1]

[1] Федеральный исследовательский центр «Информатика и управление» Российской академии наук (ФИЦ ИУ РАН)

# МЕТОДЫ ЗАМЕЩЕНИЯ ДАННЫХ В КЭШ-ПАМЯТИ

*В системах иерархической памяти разработка метода замещения данных в кэше занимает важное место. Необходимость в таком замещении объясняется ограниченным объемом кэш-памяти, что обусловливает выбор данных, которые должны быть вытеснены из кэша в главную память. Эффективные методы замещения должны вытеснять из кэша такие данные, обращение к которым в главной памяти вызвало бы наименьшие потери времени. В данной работе предлагается сравнительно недорогая реализация метода замещения, основанного на одновременных обращениях процессора к памяти, вызвавших промахи в кэше. Кроме этого, в работе анализируются два других метода и на основании этого анализа предлагаются рекомендации по применению каждого из этих методов.*

***Ключевые слова:*** *кэш-память, замещение данных в кэше, параллельная обработка промахов в кэше, устранение зависимости по данным.*

**Введение**

С развитием и совершенствованием микропроцессорных систем (МС) различие между временем выполнения команд в процессоре и временем выборки данных из памяти все время возрастает, что является тормозом для повышения производительности МС. Для снижения зависимости производительности МС от времени выборки данных из памяти используется кэш-память, время выборки из которой существенно ниже, чем время выборки из главной памяти. Однако объем кэш-памяти сравнительно невелик. Поэтому при необходимости размещения в кэше новой строки, то есть при промахе в кэше при обращении к нему, из кэша должна быть вытеснена размещенная в ней строка и замещена новой строкой. Для удаления строки кэша при необходимости помещения новой строки, как правило, используется алгоритм LRU (Least Recently Used – давно используемая строка), то есть вновь помещаемая в кэш строка занимает место давно используемой строки. Последняя вытесняется из кэша, а вновь поступившая строка занимает ее место в кэше и считается только что используемой строкой. В последние годы появилось много работ, в которых показано, что применение других методов замещения данных в кэше позволяет достичь для некоторых прикладных программ существенно большей производительности по сравнению с алгоритмом LRU.

К таким методам относятся:

- метод, основанный на учете одновременных обращений процессора к памяти, вызвавших промахи в кэше;
- метод, учитывающий наличие в процессорах средств, снижающий потери от промаха в кэше;
- метод, учитывающий длительность интервалов при повторных обращениях к одному и тому же блоку данных (строке кэша).

Для реализации любого из этих методов замещения строк в кэше необходимо обладать точной информацией о характере обращения к данным в приложении.

Однако управление кэша не обладает такой информацией. Поэтому при реализации методов замещения строк в кэше делаются предположения о характере обращения к данным в приложении на основании различной косвенной информации.

В данной работе предлагается сравнительно недорогая реализация метода замещения, основанного на учете одновременных обращений процессора к памяти, вызвавших промахи в кэше. Кроме этого, в работе кратко анализируются два остальных метода и на основании этого анализа выработаны рекомендации по применению каждого из этих методов.

**Параллельная обработка промахов**

Введем понятие обработки промаха и времени обработки промаха. Под обработкой промаха в кэше понимается процесс, состоящий из следующих этапов:

- обнаружение факта отсутствия запрашиваемых данных в кэше;
- передача запроса на следующий уровень иерархии памяти, то есть в кэш 2, кэш 3 (если имеется) или в главную память;

- выборка данных из памяти следующего уровня иерархии;
- передача выбранных данных процессору.

Время обработки промаха – это время от обращения процессора к памяти до получения процессором запрашиваемых данных.

В современных процессорах время простоя конвейера зависит не только от задержки при обращении к памяти, но и от ситуации, возникшей при обработке промахов. Большое влияние на потери времени от промаха в кэше оказывает порядок возникновения промахов в кэше. Если между последовательными промахами проходит небольшой отрезок времени, порядка 10–30 тактов, то такие промахи обрабатываются параллельно. При этом задержка от промаха в кэше будет частично скрыта за счет параллельной обработки промахов. Задержка от потери при промахе в кэше будет тем меньше, чем больше степень параллелизма обработки промахов. Под степенью параллелизма обработки промахов понимается количество одновременно обрабатываемых промахов. Если же промах возникает отдельно от других промахов, то есть следующий промах возникает через большой промежуток времени, порядка 250 и более тактов, то потеря времени из-за обработки такого промаха существенно снижает производительность. Такой промах будем называть изолированным. Изолированные промахи и промахи с низкой степенью параллелизма должны сохраняться в кэше. В противном случае, то есть при размещении их в главной памяти, промах в кэше при обращении к таким блокам приведет к существенной задержке в выполнении приложения и к потере производительности.

Эти соображения легли в основу метода замещения в кэше, учитывающего параллельность обработки промахов, – метода PM (Parallel Miss).

В данной работе предлагается сравнительно недорогая реализация алгоритма выявления параллельных промахов и вычисления степени параллелизма обработки таких промахов. Эта реализация намного дешевле по использованию дополнительных ресурсов по сравнению с предлагаемой в работе [1], в которой реализация этого алгоритма включает довольно сложную структуру регистра хранения статуса промаха (Miss Status Holding Register – MSHR). В строки этой структуры записываются промахи в кэше.

Рассмотрим примеры изолированных и параллельных промахов (рисунок).

На рисунке показаны запросы к памяти R1–R5, вызвавшие промахи в кэше, а также такты возникновения запросов и конца их обработки.

Промахи в кэше при запросах R1 и R5 – изолированные. На обработку каждого из них требуется 200 тактов. Промахи в кэше при запросах R2–R4 – параллельные. На обработку каждого из них требуется (490–250)/3=80 тактов. Из показанного здесь примера легко установить, что при увеличении количества одновременно обрабатываемых промахов, то есть при повышении степени их параллелизма, на обработку каждого из промахов требуется меньше времени. Алгоритм замещения, использующий параллелизм обработки промахов, заключается в сохранении в кэше блоков, соответствующих изолированным промахам и промахам с малой степенью параллелизма. При необходимости вытеснения блоков из кэша кандидатами на вытеснение являются блоки с наибольшей степенью параллелизма.

**Реализация алгоритма замещения, учитывающего параллельную обработку промахов**

Метод замещения в кэше, учитывающий параллельность обработки промахов, – алгоритм PM (Parallel Misses) – состоит из следующих этапов:

- разделение блоков на 2 категории в зависимости от того, какой тип промаха возникает при обращении к блоку – изолированный или параллельный;
- определение степени параллелизма при обработке промахов;
- выделение блоков с высокой степенью параллелизма как кандидатов на замещение в кэше.

Для реализации алгоритма определения степени параллелизма промахов требуется небольшая аппаратно-программная избыточность. Необходимо добавить следующие средства:

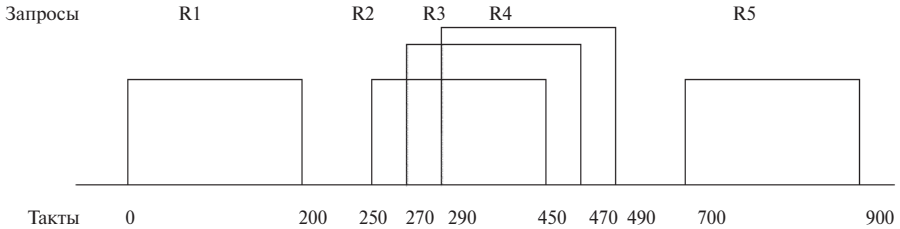


Рисунок 1. Примеры изолированных и параллельных промахов

- регистр предыдущего промаха – $R_{prev\_miss}$, хранящий номер такта появления запроса, вызвавшего предыдущий промах;
- регистр текущего промаха – $R_{current\_miss}$, хранящий номер такта появления запроса, вызвавшего текущий промах;
- счетчик параллельных промахов – $C_{paral\_miss}$;
- поле хранения значения счетчика параллельных промахов для каждого блока – $paral\_miss_n$ – поле параллельного промаха блока $n$.

Установка факта наличия параллельного промаха выполняется следующим образом. Если запрос, вызвавший текущий промах, возник раньше конца обработки запроса, вызвавшего предыдущий промах, то оба этих промаха являются параллельными. Если это условие выполнено, то счетчик параллельных промахов – *Cparal_miss* – инкрементируется. Значение счетчика параллельных промахов *Cparal_miss* передается в поля хранения этого значения соответствующих блоков.

Конец последовательности параллельных промахов определяется по двум условиям:

1. Запрос, вызвавший текущий промах, возник позже конца обработки предыдущего промаха.
2. Счетчик параллельных промахов не равен нулю.

При обнаружении конца последовательности параллельных промахов счетчик параллельных промахов обнуляется. На рис. 1 примером конца последовательности параллельных промахов может служить промах при запросе R4 – последний промах в последовательности параллельных промахов. Последующий промах при запросе R5 возникает позже конца обработки предыдущего промаха по запросу R4 и счетчик параллельных промахов не равен нулю.

Изолированный промах определяется по двум условиям:

1. Запрос, вызвавший текущий промах, возник позже конца обработки предыдущего промаха.
2. Счетчик параллельных промахов равен нулю.

Нулевое значение счетчика параллельных промахов передается в поля хранения этого значения соответствующих блоков.

В современных процессорах имеется средство для определения количества тактов после команды сброса. Так, во всех процессорах x86, начиная с Pentium, имеется 64-битный регистр, хранящий значение счетчика тактов после команды сброса. С помощью инструкции RDTSC значение счетчика тактов записывается в регистры EDX: EAX. Ниже представлен описанный выше алгоритм на языке С. $N$ – время обработки промаха в кэше в тактах. Контроллер кэша передает процессору номер такта появления запроса, вызвавшего промах. После этого процессор передает номер такта в регистр $R_{current\_miss}$.

```
R_prev_miss = 0; // обнуление регистра хранения такта предыдущего промаха
N: R_current_miss = 0; // обнуление регистра хранения такта текущего промаха
C_paral_miss = 0; // обнуление счетчика параллельных промахов
if (R_current_miss ! = 0) // появление первого промаха
{
M: (R_current_miss < (R_prev_miss + N)) // появление следующего промаха в
// последовательности параллельных промахов
{
C_paral_miss ++;
R_prev_miss = R_current_miss;
go to M // промах параллельный
}
else
if (C_paral_miss ! = 0) // конец последовательности параллельных промахов
{
paral_miss_n = C_paral_miss // передача значения счетчика параллельных
// промахов в поле хранения этого значения блока n
C_paral_miss = 0;
}
else
paral_miss_n = 0; // промах изолированный
go to N
}
```

**Метод, учитывающий наличие в процессорах средств, снижающих потери от промаха в кэше**

Причиной потери времени из-за промаха в кэше при чтении из памяти является зависимость по данным. Эта зависимость возникает в случае, если после инструкции загрузки данных следуют инструкции, использующие эти данные. Если при загрузке данных процессор обращается к кэшу и в нем происходит промах, конвейер процессора приостанавливается и продолжает работу только после получения запрашиваемых данных.

В суперскалярных процессорах имеются средства, позволяющие устранить или ослабить зависимость по данным с помощью буфера переупорядочения инструкций – ROB (reorder buffer). Устранение или ослабление зависимости по данным делает возможным скрыть хотя бы частично потерю времени из-за промаха в кэше.

Предположим, что в результате переупорядочения команд после команды загрузки появилась цепочка команд, независимых от этой команды

загрузки. Тогда при промахе в кэше конвейер будет продолжать работу, а параллельно из главной памяти будут выбираться запрашиваемые данные. Таким образом будет скрыта задержка, вызванная промахом в кэше. В таком случае запрашиваемые данные следует разместить в главной памяти. Тогда проявится эффект от переупорядочения команд. При необходимости вытеснения данных из кэша именно такие данные должны быть вытеснены в главную память. Если же такие данные будут размещены в кэше, то эффект от переупорядочения команд не проявится.

В работе [2] предлагается разделить блоки (строки) в кэше на две категории по количеству инструкций, которые может выполнить процессор за время обработки промаха при обращении к данному блоку без остановки конвейера. К первой категории относятся блоки, при обращении к которым за время обработки промаха выполняется относительно небольшое количество инструкций. Такие блоки считаются блоками с низкой стоимостью. Ко второй категории относятся блоки, при обращении к которым за время обработки промаха выполняется относительно большое количество инструкций. Такие блоки считаются блоками с высокой стоимостью. Для разделения этих двух категорий устанавливается порог стоимости, выше которого блок считается высокостоимостным, а ниже – низкостоимостным. Такой метод назван CSCR (Cost-Sensitive Cache Replacement), то есть методом, учитывающим при замещении в кэше блоков данных стоимость этих блоков.

Производительность процессора может быть повышена, если из кэша будут вытеснены блоки с высокой стоимостью, сохраняя в кэше блоки с низкой стоимостью. При обращении к блокам с высокой стоимостью, размещенным в главной памяти, повышается вероятность того, что задержка, определяемая временем обработки промаха, будет скрыта переупорядочением команд. Если же при обращении к блоку кэша промаха не возникает, то не имеет значения, к блоку какой из категорий обращается процессор.

Представленные в работе [2] данные показывают, что количество выполненных инструкций за время обработки различных промахов существенно меняется при различных промахах. Для многих приложений, с которыми проводились эксперименты, применение метода CSCR не показало заметного повышения производительности.

**Метод, учитывающий длительность интервалов при повторных обращениях к одному и тому же блоку данных (строке кэша)**

В алгоритме LRU предполагается, что между недавно используемой строкой и повторным обращением к ней будет небольшой интервал. То есть считается, что при обработке данных проявляется временная локальность. Под такой локальностью понимается небольшой интервал времени между повторными обращениями к одной и той же строке кэша. Однако при работе многих приложений при обращении к данным не проявляется временная локальность. Набор неповторяющихся данных зачастую превышает размер кэша. В работе [3] предлагается метод предсказания интервала повторного обращения к данным в кэше – RRIP (Re-Reference Interval Prediction).

Прежде всего, рассмотрим некоторые типичные последовательности обращения к данным [3].

$$(a1, a2, ..., ak-1, ak, ak, ak-1, ..., a2, a1)\ N \qquad (1)$$

$k <= C$, где $C$ – количество строк в кэш

$$(a1, a2, ..., ak)\ N. \qquad (2)$$

$K$ принимает любое значение

$$(a1, a2, a3, a4, ...\ ak) \qquad (3)$$

$k = \infty$

$$[(a1, ..., ak)A\ (b1, b2, ..., bm)]\ N \qquad (4)$$

$k < C$ и $m > C$.

Во всех примерах через $N$ и $A$ обозначено число повторений последовательностей. $ai$ и $bi$ – уникальные номера блоков данных (каждый блок эквивалентен строке кэша).

Последовательность (1) – наиболее подходящая для алгоритма замещения LRU. Пусть $k = C$, тогда после начального заполнения кэша в нем будет следующее размещение строк: $ak$, $ak$, $ak-1$, ..., $a2$, $a1$. При всех повторениях данной последовательности не произойдет ни одного промаха, так как все строки уже имеются в кэше.

Последовательность (2) представляет циклическое обращение к данным длиною $k$. При $k > C$ происходит перегрузка кэша. Это явление называется трэшем. При этом в алгоритме замещения LRU любое обращение в кэш приводит к промахам в нем (misses), так как после заполнения всего кэша при поступлении в него новых строк вытесняются блоки $a1, a2, ...$ как давно использованные. Последовательность (2) повторяется $N$ раз, каждое повторное обращение к блокам $a1, a2, ...$ приводит к промаху в кэше.

Последовательность (3) – бесконечный поток уникальных блоков данных. При таком обращении к данным никакой из алгоритмов замещения не может уменьшить количество промахов.

Последовательность (4) – смешанный поток обращений, в котором сочетаются два потока: с коротким и длинным интервалами повторного обращения к данным. Длинный интервал представлен на сером фоне. Первой представлена последовательность длиною $k$, повторяющаяся $A$ раз, второй представлена последовательность длиною $m$, повторяющаяся вместе с первой последовательностью $N$ раз. Причем длина первой последовательности меньше объема кэша ($k < C$), а длина второй последовательности больше объема кэша ($m > C$). В дальнейшем для краткости данные длиною $k$ будем называть активным набором, а данные длиною $m$ – сканом.

В методе RRIP каждому блоку в кэше для предсказания величины интервала между повторными обращениями к данным присваивается $M$ бит для сохранения $2^M$ возможных значений интервалов между повторными обращениями. Для подсчета значений интервалов используются $M$-битные счетчики с насыщением. Значения этих счетчиков хранятся в регистрах RRPV (Rereference Prediction Value).

Задачей RRIP является предотвращение размещения в кэше блоков с длинными интервалами повторения обращения к ним, чтобы они не «загрязняли» кэш. При помещении в кэше нового блока RRIP присваивает ему увеличенное значение RRPV = $2^M - 2$, то есть промежуточное значение, приближенное к наибольшему интервалу – $2^M - 1$. Итак, предположим, что помещаемый в кэш блок принадлежит к скану. Если же окажется, что вновь помещенный блок относится к активному набору, то RRIP изменяет величину интервала между повторными обращениями к блоку, уменьшая RRPV блока.

При промахе в кэше RRIP выбирает в качестве кандидата на вытеснение блок с наибольшим значением RRPV, равным $2^M - 1$. Если блок с RRPV, равным $2^M - 1$, отсутствует в кэше, то RRIP последовательно инкрементирует RRPV всех блоков до тех пор, пока значение RRPV какого-либо блока не станет равным $2^M - 1$.

При попадании в кэш, то есть при нахождении запрашиваемого блока в кэше, появляется возможность динамически повысить точность предсказания повторного обращения к данным. Предполагается, что при обращении к блоку, уже размещенному в кэше (попадание в кэш), обращение к такому блоку будет повторено в ближайшее время. Регистр RRPV такого блока обнуляется. При этом повышается приоритет на вытеснение блоков, к которым не было обращений.

**Сравнение эффективности методов замещения данных в кэше**

В таблице приведено сравнение по производительности при использовании методов замещения PM, CSCR и RRIP с методом LRU. Источником этих данных являются опубликованные в работе [1] результаты моделирования микропроцессорных систем с различными приложениями.

**Заключение**

По выбранным приложениям эффективность рассматриваемых методов замещения данных в кэше примерно одинакова. Предпочтение следует отдать PM и RRIP, так как для большого числа приложений применение метода CSCR не показало заметного повышения производительности. По дополнительным ресурсам и простоте реализации предпочтение следует отдать методу PM, к тому же предложенная в данной работе его реализация существеннее проще, чем в работе [1].

Таблица. Увеличение производительности по сравнению с алгоритмом LRU

| Приложения | Методы замещения | | |
|---|---|---|---|
| | PM | CSCR* | RRIP |
| mcf | 1,4 | Данные отсутствуют | 1,4 |
| soplex | 1,05 | Данные отсутствуют | 1,04 |
| libquantum | 1,03 | 1,02 | 1 |
| lbm | 0,97 | 1 | 0,97 |
| omnetpp | 1,36 | 1,2 | 1,17 |
| astar | 0,95 | 1,0 | 0,94 |
| splinx3 | 1,04 | 1,07 | 1,06 |
| hmmer | 1,05 | 1,17 | 1,03 |
| Средние значения | 1,1 | 1,07 | 1,08 |

* Для метода CSCR в таблице приведены только приложения, при работе которых производительность не уменьшалась по сравнению с методом LRU.

## СПИСОК ЛИТЕРАТУРЫ

1. Li L., Lu J., Cheng X. Retention Benefit Based Intelligent Cache Replacement. Journal of Computer Science and Technology Nov. 2014, pp. 947–961.
2. Kharbutli M., Sheikh R.. LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm. IEEE Trans. on Computers, vol. 63, issue 8, Aug. 2014, pp. 1975–1987.
3. Jaleel A., Theobald K., Steely S., Emer J. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP) // ACM SIGARCH Computer Architecture News – ISCA ‘10, vol. 38, issue 3, June 2010, pp. 60–71.

## ИНФОРМАЦИЯ ОБ АВТОРЕ

**Шмейлин Борис Захарьевич,** к.т.н., старший научный сотрудник, ФИЦ ИУ РАН, Москва, ул. Вавилова, д. 44, к. 2, тел.: +7 (499) 135-42-25 (доб. 2413), +7 (903) 108-23-55, e-mail: shmeilin@mail.ru.

*For citation: Shmeylin B. Data replacement methods in the cache-memory. Voprosy radioelektroniki, 2017, no. 2, pp. 43–48.*

**B. Shmeylin**

## DATA REPLACEMENT METHODS IN THE CACHE-MEMORY

In a hierarchical memory system development data in the cache replacement method takes an important place. The need for such replacement is explained by the limited capacity of the cache memory that causes the selection of data to be pushed from the cache to the main memory. The efficient methods for replacement must push from the cache such data are accessed in the main memory would cause the least loss of time. In this paper is proposed a relatively inexpensive implementation of the replacement method based on the simultaneous requests to memory that caused the cache misses. In addition, the paper analyses the other replacement methods and on the basis of this analysis provides recommendations on the application of each of these methods.

**Keywords:** cache-memory, replacement the data in the cache, misses, parallel misses processing, eliminating data dependency.

## REFERENCES

1. Li L., Lu J., Cheng X. Retention Benefit Based Intelligent Cache Replacement. Journal of Computer Science and Technology Nov. 2014, pp. 947–961.
2. Kharbutli M., Sheikh R. LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm. IEEE Trans. on Computers vol. 63, issue 8, Aug. 2014, pp. 1975–1987.
3. Jaleel A., Theobald K., Steely S., Emer J. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP) // ACM SIGARCH Computer Architecture News – ISCA ‘10, vol. 38, issue 3, June 2010, pp. 60–71.

## AUTHOR

**Shmeylin Boris,** PhD, FIC IU RAN, 44, k. 2, Vavilova st., Moscow, tel.: +7 (499) 135-42-25 (2413), +7 (903) 108-23-55, e-mail: shmeilin@mail.ru.