Обнаружение и оценка количества промахов когерентности на основе вероятностной модели

Е.А. Велесевич <evel@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Ложное разделение кэша возникает, когда нити, параллельно выполняющиеся на разных ядрах, поочередно обновляют разные переменные, попадающие в одну строку кэша, что приводит к устареванию информации о текущем состоянии памяти в кэше ядра, используемом первой нитью приложения, и необходимости нити ждать, пока информация в кэше обновится. В статье для оценки количества промахов предлагается использовать инструментацию кода и постобработку ее результатов: по наблюденным промахам кэша в трассе обращений к памяти с временными метками для каждой нити вычисляется вероятность того, что во время обращения, выбранного с некоторой заранее определенной вероятностью, и следующего обращения к этой строке в этой же нити была запись в ту же строку в другой нити. Трассировщик программы реализован как проход в открытом компиляторе GCC, добавляющий специальные инструкции перед каждым обращением к памяти и выполняющийся после всех оптимизирующих проходов, благодаря чему становится возможной оценка эффективности использования кеша в оптимизированных приложениях. Анализатор реализован в виде отдельного приложения, которому подается на вход сгенерированные на тестовом наборе данных трассы обращений к памяти, имевших место в нитях анализируемого приложения. Замедление работы программы при трассировке для проверенных экспериментально тестовых приложений составляет примерно 10 раз, при этом оно зависит от вероятности выборки, которая в свою очередь выбирается в зависимости от характера и времени работы приложения, но практически не зависит от длины кэш-линии.

Ключевые слова: ложное разделение кэша; GCC; инструментация кода.

DOI: 10.15514/ISPRAS-2015-27(4)-3

Для цитирования: Велесевич Е.А. Обнаружение и оценка количества промахов когерентности на основе вероятностной модели. Труды ИСП РАН, том 27, вып. 4, 2015 г., стр. 39-48. DOI: 10.15514/ISPRAS-2015-27(4)-3.

1. Введение

Кэширование часто используемых данных из оперативной памяти является одним из ключевых методов обеспечения высокой производительности современных процессоров. Кэши не является общими ресурсами: каждое процессорное ядро, как правило, имеет свой кэш первого уровня; в многопроцессорных системах каждый физический процессор имеет свою иерархию кэшей. Кэш хранит множество выровненных участков оперативной памяти, называемых строками. Один и тот же участок оперативной памяти может одновременно находиться в кэшах разных процессоров; когда один из процессоров модифицирует эти данные, необходимо обновить или удалить их из кэша другого процессора. Эта задача решается с использованием протокола согласования кэшей (cache coherency protocol).

Поскольку модификация даже одного байта приводит к обновлению целой строки кэша, могут возникать ситуации, когда нити, параллельно выполняющиеся на разных ядрах, поочередно обновляют разные переменные, попадающие в одну строку кэша. В таких случаях обновление на одном процессоре будет приводить к вытеснению соответствующей строки из кэша другого процессора и сериализовывать выполнение нитей, как если бы они обновляли одну и ту же переменную. Такое поведение известно как ложное разделение (false sharing) и ухудшает производительность параллельных программ. Соответственно, желательно иметь инструменты для анализа эффективности использования кэша, в том числе поиска ситуаций ложного разделения.

Далее в статье в разделе 2 описывается предлагаемый метод поиска ситуаций борьбы за кэш. В разделе 3 предлагается его реализация в инструменте на основе открытого компилятора GCC. В разделе 4 содержатся некоторые экспериментальные результаты. Раздел 5 завершает статью.

2. Метод поиска ошибок борьбы за кэш

Для оценки количества промахов доступа к кэшу можно использовать инструментацию кода. Основная идея заключается в том, чтобы собрать прореженную трассу доступов к памяти, включая расстояния повторного использования, и затем оценить количество промахов исходя из собранной трассы и некоторой теоретической модели кэша. Хагерстен [1] предлагает вероятностную модель для оценки эффективности использования кэша со случайным вытеснением на основе расстояния повторного использования строк кэша. Недостатком его подхода для поиска ситуаций ложного разделения в многопоточных программах является необходимость дополнительной синхронизации при сборе трассы. Нашей целью будет обойти эту необходимость, так как она может существенно замедлить выполнение программы.

Пусть L — количество строк в кэше; тогда вероятность того, что строка, находящаяся в кэше, уже не будет в нем после n промахов, равна:

$$f(n) = 1 - \left(1 - \frac{1}{L}\right)^n$$

Пусть $r(i)$ – вероятность промаха при i -ом обращении, а $A(i)$ – значение расстояния повторного использования для i -го обращения. Тогда количество промахов кэша между i -м и (i-$A(i)$-1)-м обращениями можно оценить как:

$$\sum_{j=i-A(i)}^{i-1} r(j)$$

Тогда вероятность того, что на момент обращения i искомая строка не будет находиться в кэше $f\left(\sum_{j=i-A(i)}^{i-1} r(j)\right)$. Но в тоже время, вероятность промаха исходно равна $r(i)$, т.е.:

$$r(i) = f\left(\sum_{j=i-A(i)}^{i-1} r(j)\right)$$

Предположив, что вероятность промаха постоянна, т.е. $r(i) = R$ для всех i (на некотором участке трассы программы), Хагерстен получает основное уравнение своей модели:

$$R * N = N_{cold} + \sum_{i=1}^{N_{nocold}} f(A(i) * R) \qquad (1)$$

В многопоточном случае модели Хагерстена запись в трассе помимо строки кэша и расстояния повторного использования содержит информацию о том, была ли запись в эту строку в другой нити приложения между первым и вторым обращением в первой нити. Такой подход к сбору информации о записях в другой нити требует синхронизации. Но возможен и другой подход — вычисление вероятности того, что во время обращения, записанного в трассу, и следующего обращения к этой строке в этой же нити, была запись в ту же строку в другой нити (на основе трасс обращений к памяти с временными метками для каждой нити приложения). Эта вероятность и будет искомым процентом промахов когерентности.

Для решения поставленной задачи требуется вычислить априорную вероятность того, что если промах когерентности произошел на обращении, записанном в трассе, то запись, благодаря которой произошел промах, была записана в трассу другой нити (таких записей может быть несколько). Пусть N_A – количество записей в кэш-строку A в исследуемом участке трассы второй нити, N – количество всех записей в участке, P – вероятность выборки

обращения для записи в трассу, T – продолжительность исследуемого участка, T_A – время между первым и вторым доступом к A в первой нити.

Тогда количество записей в участке есть $N_w = N/P$, среднее время между записями есть $T_w = T/N_W$, количество записей во второй нити между первым и вторым обращением в первой нити есть $N_{A_g} = T_A/T_w$.

А вероятность того, что между первым и вторым доступом будет k записей в ту же строку в другой нити, есть:

$$P_{A_k} = C_k^{N_{A_g}} P_A^K (1 - P_A)^{N_{A_g} - k}, \text{ где } P_A = N_A/N. \qquad (2)$$

Тогда вероятность, что в трассе будет обращение, благодаря которому произошел промах (хотя бы одно из них):

$$P_{A_d} = 1 - \sum_{k=1}^{N_{A_g}} P_{A_k} (1 - P)^k$$

А вероятность, что промах произошел, независимо от того, попал он в трассу или нет:

$$P_{A_m} = 1 - (1 - P_A)^{N_{A_g}}$$

Однако, нам известна статистическая вероятность попадания промаха в трассу:

$$\bar{P}_{A_d} = \frac{\bar{N}_{A_d}}{N},$$

где $\bar{N}_{A_d}$– количество замеченных промахов.

Тогда условная вероятность попадания промаха в трассу при наличии такого промаха:

$$\frac{\bar{P}_{A_d}}{P} = \frac{P_{A_d}}{P_{A_m}}$$

Из этого уравнения можно получить P – искомую вероятность промаха когерентности, или же процент обращений к памяти, повлекших промах когерентности.

Для достаточно точного определения $\bar{P}_{A_d}$ требуется намного более частая выборка обращений для записи в трассу, так как вероятность того, что между выбранными с вероятностью P обращениями в другой нити была выбрана запись, влекущая промах когерентности, тоже равна P, т.е. $\bar{P}_{A_d}$ порядка P^2. Поэтому P следует выбирать как квадратный корень из вероятности в модели Хагерстена.

3. Реализация инструмента поиска ситуаций борьбы за кэш

На основе вышеизложенной модели была реализована программа, оценивающая использования кэшей приложениями с несколькими исполняемыми потоками. Программа состоит из двух частей: трассировщик снимает трассы приложения для последующей обработки, анализатор вычисляет количество промахов кэша (промахи объема и промахи когерентности), а также информирует о месте в программе в которой произошел промах.

Трассировщик реализован в виде отдельного прохода в компиляторе GCC [2], вставляющего в необходимых местах вызовы функции (реализованной в отдельной библиотеке), осуществляющей выборку доступов в память для записи в трассу. Для сборки с трассировкой модернизированному компилятору нужно передать необходимые опции и при необходимости указать путь к библиотеке трассировки. Запуск трассируемого приложения происходит в обычном режиме. По окончании работы приложения в папке с исполняемым файлом появится несколько файлов трасс (по одной для каждой нити приложения), который необходимо будет передать анализатору.

Трассировку можно настраивать, задавая вероятность выборки доступа в память и длинны кэш-линии путем установки переменных окружения. Отметим необходимость задания длины кэш-линии уже в трассировщике, так как ему необходимо определять, в какую кэш-линию происходит доступ для решения о внесении его в трассу. В целом замедление работы программы составляет ~10 раз, но оно зависит от вероятности выборки (естественно, чем меньше доступов мы будем записывать, тем меньше замедление) и практически не зависит от длины кэш-линии.

Анализатор реализован в виде отдельного приложения, которому подаются на вход трасса (состоящая из нескольких подтрасс) анализируемого приложения. Анализатор работает в несколько стадий:

1. Вычисление распределения доступов к памяти по расстояния повторного использования и пометка промахов когерентности, попавших в трассу (и сообщение о них пользователю).
2. Численное решение модели Хагерстена для оценки промахов объема. Уравнение (1) решается методом простых итераций для поиска корня на интервале (0,1).
3. Вычисление априорной вероятности попадания промахов когерентности в трассу.
4. Вычисления на основе сравнения априорной и статистической вероятности попадания промахов в трассу.

В качестве параметра анализатору можно передать объем кэша.

4. Экспериментальные результаты

Для первичной проверки модели расчета промахов когерентности была использована тестовая трасса, состоящая их трех подтрасс (т.е. в приложении было три нити) с обращениями в память упорядоченными таким образом.

t1	t2	t3	t1	t2	t3	t1	t2	t3	t1

т.е. за обращением в первой нити, следовало обращений во второй нити, потом в третьей и снова в первой. Обращения происходили в массив размером с кэш, т.е. промахи объема почти отсутствовали. Естественно, трасса доступов к памяти была прорежена случайным образом, так что в итоге могло получаться примерно такое:

t1	t1	t1	t1	t2	t3	t2	t3	t1	t1

Можно вычислить априорную вероятность промахов когерентности следующим образом. Пусть b – вероятность обращения по определенному адресу (обратно пропорциональна размеру массива (кэша)), тогда вероятность, что при обращении по адресу A в нити 1 следующее обращение будет не в нити 1:

$$\big(1-(1-b)(1-b)\big)+(1-b)(1-b)(1-b)(\big(1-(1-b)(1-b)\big)+\cdots=$$

$$=(1-(1-b)^2)\frac{1}{1-(1-b)^3}=(2-b)/(b^2-3b+3)$$

$$\lim_{b\to 0}\frac{2-b}{b^2-3b+3}=\frac{2}{3}$$

Разработанный анализатор выдает правильные результаты для тестового примера (>66.(6)%, так как массив все же не бесконечный):

16мб кэш	нить 0	нить 1	нить 2
Ошибки когерентности	67.1%	66.8%	66.8%

Далее, было проведено тестирование инструмента на нескольких простых приложениях из пакета coreutils. Промахи кэша в программе sort (64 байт в

кэш-линии) представлены в таблице 1 (промахи объема) и таблице 2 (промахи когерентности).

Таблица 1. Промахи объема в программе sort.

Размер кэша	нить 0	нить 1	нить 2	нить 3	нить 4	нить 5	нить 6	нить 7
16мб	6.5%	2.7%	2.3%	2.4%	3.2%	3.0%	2.5%	3.0%
8мб	7.3%	3.1%	2.5%	2.6%	3.7%	3.3%	2.8%	3.4%
2мб	9.5%	4.7%	4.3%	4.5%	4.4%	4.4%	4.5%	4.8%

Таблица 2. Промахи когерентности в программе sort.

Размер кэша	нить 0	нить 1	нить 2	нить 3	нить 4	нить 5	нить 6	нить 7
16мб	0.5%	0.7%	0.7%	1.5%	0.2%	0.6%	1.0%	0.5%
8мб	0.5%	0.7%	0.7%	1.4%	0.3%	0.7%	1.0%	0.5%
2мб	0.5%	0.7%	0.7%	1.4%	0.3%	0.7%	1.0%	0.5%

Как можно видеть, промахи когерентности не слишком зависят от объема кэша. Чем меньше объем кэша, тем меньше вероятность, что другая нить испортит его содержимое. Но, как мы видим, объем кэша не очень сильно влияет на ошибки объема в приложении sort, поэтому мы и не видим уменьшения промахов когерентности в таблице.

Было получено несколько сообщений о ложном разделении, например:

```
    False sharing is detected:

    Previous access at ../../coreutils/coreutils-
8.16/src/sort.c:3080
```

```
    Conflicting access at ../../coreutils/coreutils-
8.16/src/sort.c:3375

    Recent access(FS) at ../../coreutils/coreutils-
8.16/src/sort.c:2673
```

В данном случае ложное разделение происходит при слиянии отсортированных участков входных данных, поэтому оно не представляет собой проблемы, несмотря на корректное сообщение инструмента, и избавляться от него не нужно.

5. Заключение

В статье описан метод поиска ситуаций борьбы за кэш и разработанный на его основе инструмент, состоящий из использующего компилятор GCC трассировщика и анализатора собранных трасс. Предложенная модель работы программы с памятью считает постоянной вероятность обращения к памяти в пределах наблюдаемого участка трассы для данной нити, что является достаточно сильным предположением, и в связи с ним часть ситуаций ложного разделения может быть потеряна инструментом. С другой стороны, наиболее вероятные ситуации разделения (а, следовательно, и наиболее нуждающиеся в исправлении) имеют больше шансов попасть в пределы наблюдаемого участка и быть обнаруженными. Поэтому можно утверждать, что ситуации ложного разделения, недостаточно проявляющиеся для того, чтобы быть обнаруженными инструментом, и не нуждаются в исправлении. Требуются дополнительные исследования на больших приложениях для ответа на вопрос о том, нужно ли расширение предложенной модели до учета переменной вероятности обращения к памяти.

Список литературы

[1]. Erik Berg, Håkan Zeffer and Erik Hagersten, A Statistical Multiprocessor Cache Model, In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006

[2]. E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis, Technical report 2003-058, Department of Information Technology,Uppsala University, November 2003.

[3]. E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In Proceedings of International Symposium on Performance Analysis of Systems And Software, March 2004

[4]. S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In SIGMETRICS ’93, pages 146–157. ACM Press, 1993.

[5]. J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for Application-Oriented Performance Tuning. In Proceedings of 15th ACM International Conference on Supercomputing, Italy, June 2001.

[6]. R.A. Uhlig and T.N. Mudge, Trace-driven memory simulation: A survey, ACM Computing Surveys, 29 (2), 1997, 128–170.

[7]. Stunkel, C. and Fuchs, W. TRAPEDS: producing traces for multicomputers via execution-driven simulation. In Proceedings of the 1989 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Berkeley, CA, ACM, 70-78, 1989.
[8]. F. Rawson. Mempower: A simple memory power analysis tool set. Technical report, IBM Austin Research Laboratory, 2004
[9]. X. Gao, B. Simon, and A. Snavely, ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording, Workshop on Binary Instrumentation and Applications, St. Louis, Mo. Sept. 2005.
[10]. Erik Berg, Methods for Run Time Analysis of Data Locality, Licentiate Thesis 2003-015, Department of Information Technology, Uppsala University, December 2003.
[11]. GCC, the GNU Compiler Collection. https://gcc.gnu.org.

Evaluating a Number of Cache Coherency Misses Based on a Statistical Model

Evgeny Velesevich <evel@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., 109004, Moscow, Russia

Annotation. False cache sharing happens when different threads executing in parallel on distinct processor cores simultaneously update the variables that reside in the same cache line. This results in the invalidation of the current memory state data that is saved in the cache utilized by the first thread's core, and also in the necessity to stall for the second thread on the memory state data update.
We suggest in this paper to evaluate the number of cache misses using code instrumentation and post-mortem trace analysis: the probability of the false sharing cache miss (defined as a memory write issued by one thread between two consecutive memory accesses issued by another thread) is calculated based on the gathered event trace, where each observed event is a memory access with a timestamp. The tracer tool is implemented as a GCC compiler pass inserting necessary tracing instructions before each memory access. The pass is scheduled after all other optimization passes that allow to use the tracer for optimized code. The post-mortem analyzer is a separate application that gets the trace collection gathered on a sample application input data as its own input. Program slowdown in our approach is ~10 times, and it is dependent on a sampling probability but it does not depend on a cache line size.

Keywords: false cache sharing; GCC compiler; code instrumentation

DOI: 10.15514/ISPRAS-2015-27(4)-3

For citation: Velesevich Evgeny. Evaluating a Number of Cache Coherency Misses Based on a Statistical Model. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 4, 2015, pp. 39-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-3.

References.

[1]. Erik Berg, Håkan Zeffer and Erik Hagersten, A Statistical Multiprocessor Cache Model, In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006

[2]. E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis, Technical report 2003-058, Department of Information Technology,Uppsala University, November 2003.

[3]. E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In Proceedings of International Symposium on Performance Analysis of Systems And Software, March 2004

[4]. S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In SIGMETRICS '93, pages 146–157. ACM Press, 1993.

[5]. J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for Application-Oriented Performance Tuning. In Proceedings of 15th ACM International Conference on Supercomputing, Italy, June 2001.

[6]. R.A. Uhlig and T.N. Mudge, Trace-driven memory simulation: A survey, ACM Computing Surveys, 29 (2), 1997, 128–170.

[7]. Stunkel, C. and Fuchs, W. TRAPEDS: producing traces for multicomputers via execution-driven simulation. In Proceedings of the 1989 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Berkeley, CA, ACM, 70-78, 1989.

[8]. F. Rawson. Mempower: A simple memory power analysis tool set. Technical report, IBM Austin Research Laboratory, 2004

[9]. X. Gao, B. Simon, and A. Snavely, ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording, Workshop on Binary Instrumentation and Applications, St. Louis, Mo. Sept. 2005.

[10]. Erik Berg, Methods for Run Time Analysis of Data Locality, Licentiate Thesis 2003-015, Department of Information Technology, Uppsala University, December 2003.

[11]. GCC, the GNU Compiler Collection. https://gcc.gnu.org.