МЕТОД ПОКРЫТИЙ ДЛЯ ОЦЕНКИ ЛОКАЛЬНОСТИ ИСПОЛЬЗОВАНИЯ ДАННЫХ В ПРОГРАММАХ

Вад.В. Воеводин, П.А. Швец

1. Введение

Одним из главных факторов, влияющих на время выполнения программ, является эффективность работы с памятью. Причина этого — так называемая проблема «стены памяти» [1], удовлетворительного решения для которой до сих пор не предложено. Иерархия памяти в компьютерах устроена чрезвычайно сложно, и время получения требуемых данных зависит не только от свойств оперативной памяти или кэш-памяти различных уровней, но и от низкоуровневых механизмов вроде аппаратного префетчера, и от TLB-буфера, и от многих причин. Все это приводит к тому, что в большинстве случаев подсистема памяти используется далеко не самым оптимальным образом, что и приводит к очень низкой эффективности работы программ [2].

Уже давно подмечено, что одним из главных свойств взаимодействия программ с памятью является локальность использования данных. Различают два типа локальности — пространственную и временну́ю [3]. Пространственная локальность данных отражает среднее расстояние по памяти между несколькими последовательными обращениями в память, в то время как временна́я локальность показывает частоту обращений по одному и тому же адресу. Чем выше локальность использования данных в программе, тем, как правило, выше и эффективность выполнения самой программ, поскольку верхние уровни в иерархии памяти используются чаще.

Цель данной работы заключается в исследовании эффективности взаимодействия программ с памятью на основе исследования свойств локальности. На основе анализа потока обращений программ в память [4] предлагается простая характеристика для оценки их пространственной и временно́й локальности. Ключевым моментом является именно простота получения и анализа характеристики, поскольку такой подход позволит легко оценивать степень эффективности работы программ с памятью, а значит и исследовать эффективность их выполнения в целом. Использование систем эмуляции работы памяти (вроде Threadspotter [5]) или сложных инструментов анализа (например, Valgrind [6]), во многих случаях позволит получать более точную и полную информацию, однако недостатком таких систем является сложность их использования: для правильной интерпретации получаемых с их помощью данных, как правило, требуются специальные знания.

Следует упомянуть и еще два важных момента, которые определяют важность разработки подходов к анализу степени локальности данных в программах. Во-первых, поскольку иерархия памяти присутствует во всех современных компьютерах, то, проводя анализ степени локальности данных конкретного приложения, мы исследуем эффективность его взаимодействия с памятью практически для всех классов вычислительных систем, что важно для сохранения эффективности при переносе приложения с платформы на платформу. Во-вторых, степень локальности данных в программах, в конечном итоге, определяется лежащими в их основе алгоритмами, и это дает путь к оценке перспективности их использования в будущем.

2. Исследование потока обращений в память

В рамках данной работы для исследования свойств локальности данных в программах предлагается подход на основе анализа потока обращений в память (или профиля обращений в память). Под потоком обращений будем понимать последовательность используемых в программе переменных, расположенных в этой последовательности в том порядке, в котором происходят обращения к этим переменным во время работы самой программы. Обозначим поток обращений

$$A = \{A1, A2, \ldots, AN\},$$

где N – это общее число всех обращений к переменным. Ai обозначает адрес виртуальной памяти, где расположена запрашиваемая переменная; номер i элемента Ai указывает, каким по счету является данное обращение от начала выполнения программы. На данный момент мы рассматриваем только обращения к массивам как самому распространенному типу структуры данных, и, кроме того, рассмотрение ограничивается последовательными программами на языке C/C++. Однако оба этих ограничения не являются принципиальными, и мы планируем в дальнейшем их устранить.

Пример профиля обращений приведен на рис. 1. Ось абсцисс — это порядковый номер обращения в память: чем больше номер, тем позже произошло данное обращение в процессе выполнения программы. По оси ординат отложен виртуальный адрес памяти, к которому произошло обращение. Каждая точка профиля отвечает отдельному обращению в память. Выделенная на рисунке область 1 (как и остальные, аналогичные ей области) обладает хорошей пространственной и временно́й локальностью (иногда вместо словосочетания ”высокая степень локальности” мы будет использовать более короткий вариант ”хорошая локальность”). Выделенная на рисунке область обладает хорошей пространственной локальностью, поскольку используются только соседние по памяти данные, и хорошей временно́й локальностью, т. к. повторные обращения к этим данным происходят часто. Область 2 профиля также обладает хорошей пространственной локальностью,

поскольку последовательные обращения в память происходят к соседним элементам, однако в этой области временна́я локальность плохая, поскольку повторных обращений к одним и тем же данным просто нет.

Для получения профиля обращений мы выполняем инструментацию исходного текста программы с помощью несложной процедуры. Для этого необходимо заменить тип массива на реализованный C++ класс, который будет вести себя также, как и исходный массив, но при любом обращении к элементу массива (вызове оператора []) будет также производить и запись в отдельный лог-файл. Поскольку мы собираем поток виртуальных обращений, запись по всем интересующим нас массивам ведется в единый лог. Чтобы уменьшить размер результирующего лог-файла и сделать сам профиль более информативным, предусмотрена возможность включать/выключать запись, чтобы не логгировать служебные или не интересующие нас фрагменты программы (к таким можно отнести, например, генерацию начальных массивов и сохранение/вывод результата).

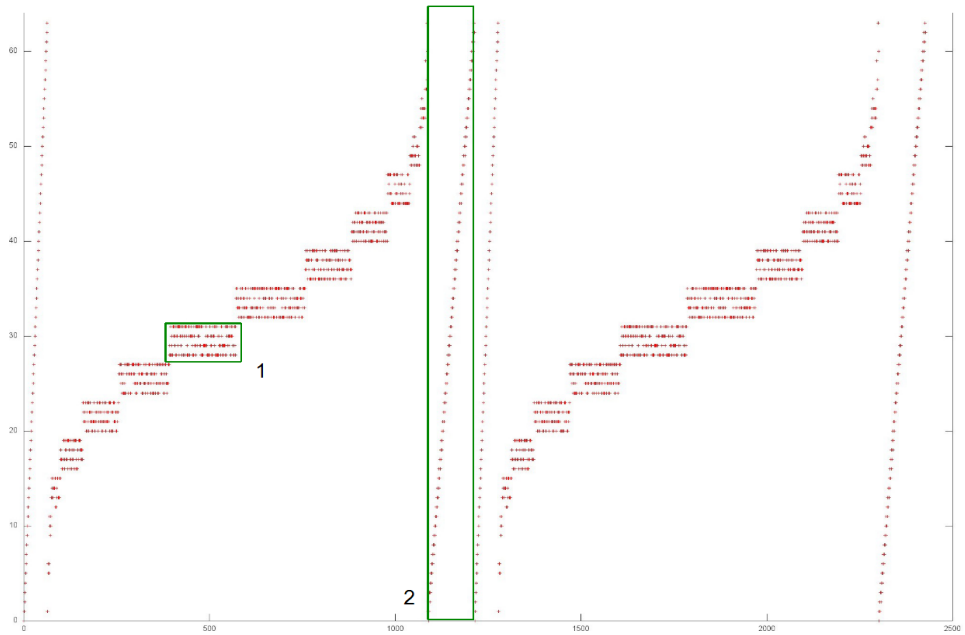

Рис.1. Пример профиля обращений в память

Важно понимать, что поток обращений, определяемый исходным текстом программы, не всегда совпадает с реальным потоком обращений, возникающим в ходе ее выполнения, поскольку реальный поток получается после множества преобразований и оптимизации текста программы компилятором. По данному преобразованному потоку в большинстве случаев можно получить более адекватную информацию, отражающую свойства заложенного в программу алгоритма. Вместе с этим, поскольку точный набор преобразований текста программы компилятором не известен, сложнее устанавливать связь между исходным текстом программы и самим профилем.

3. Покрытие как основа для оценки локальности

Итак, наша цель заключается в поиске одной или нескольких простых характеристик, которые помогут достаточно точно оценивать свойство локальности полученного потока обращений. В результате проведенного исследования была предложен следующий подход.

Рассмотрим первые N обращений в потоке, и указанную совокупность из N последовательных обращений далее будем называть «окно». Разобьем ось ординат на отрезки длины K:

$$[X .. X+K-1],\ [X+K .. X+2K-1], \ldots, [Y-K+1 .. Y],$$

где [X, Y] — это отрезок адресов от X до Y виртуальной памяти, выделенный под рассматриваемые массивы. Таким образом, профиль обращений делится на прямоугольники размером KxN. Если предположить, что значения K и N подобраны правильно, то ”физический смысл” такого прямоугольника заключается в том, что любой набор точек, попавших в один прямоугольник, всегда будет обладать хорошей как пространственной, так и временно́й локальностью. На практике, слова ”K и N подобраны правильно” означают согласование значений K и N с параметрами подсистемы памяти конкретной вычислительной системы и, прежде всего, с параметрами кэш-памяти.

Теперь подсчитаем число прямоугольников, в которых оказалась хотя бы одна точка профиля. Ясно, чем меньше таких прямоугольников, тем больше точек в среднем в одном прямоугольнике, тем в общем случае лучше и локальность. На рис. 2 показаны разные с точки зрения локальности фрагменты профилей и их покрытия системой прямоугольников. Окно 1 является фрагментом профиля с последовательными

обращениями — здесь наблюдается хорошая локальность, лишь один прямоугольник является непустым. В окне 2 представлен фрагмент профиля также с последовательными обращениями, но, в отличие от первого случая, идущие с некоторым шагом. В этом случае локальность будет хуже, чем у первого варианта: в окне занято 2 прямоугольника. Окно 3 является фрагментом профиля со случайной последовательностью обращений: здесь явно наблюдается плохая локальность, занято 3 прямоугольника.

Сдвинем окно на одну точку, т. е. рассмотрим обращения с 1 по N+1, и снова посчитаем число непустых прямоугольников. Далее выполним аналогичные действия по всему потоку обращений. Назовем покрытием полученное множество непустых прямоугольников для каждого окна.

Теперь посчитаем число непустых прямоугольников для каждого окна, в результате чего получим последовательность чисел, каждое из которых, в общих чертах, характеризует локальность в некоторой точке работы программы. Обозначим через Cvg среднее значение по всей последовательности. По сути это значение показывает, сколько прямоугольников необходимо для покрытия профиля.

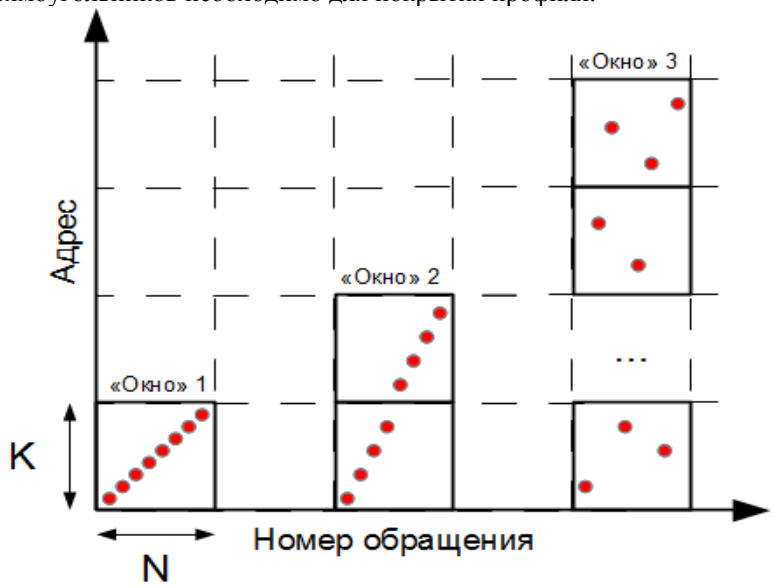

Рис.2. Пример покрытия различных профилей

Безусловно, достаточно серьезный вопрос заключается в подборе значений параметров K и N. Желательно взять для них максимальные значения, при которых в рамках прямоугольника все еще сохраняется свойство хорошей локальности. На данный момент в своих исследованиях мы использовали значение K=64. Такое значение выбрано из тех соображений, что при запросе данных из памяти в кэш-память 1-го уровня всегда подтягивается не только само это значение, но и вся кэш-строка: практически во всех современных процессорах размер кэш-строки равен 64 байтам. Соответственно, если взять K=64, то мы можем быть уверены, что если произошел запрос данных по некоторому виртуальному адресу, то подтянутся также все остальные данные, соответствующие тому же прямоугольнику.

Поскольку мы считаем, что в рамках прямоугольника и временна́я локальность также должна быть хорошей, это означает, что если было запрошено значение T по некоторому адресу A, то через N обращений значение T с высокой вероятностью должно оставаться в кэш-памяти. С учетом этого требования, в настоящий момент эвристическим методом выбрано значение N=128, однако в дальнейшем это значение может быть пересмотрено.

4. Экспериментальные исследования

Все эксперименты проводились на последовательных реализациях широко распространенных программ:

- RandomAccess и STREAM из набора тестов HPCC [7],
- последовательная версия теста Linpack [8],
- БПФ над вещественными числами,
- классическое (не блочное) перемножение матриц,
- операция триада.

Тест STREAM представлен в 4 вариантах [9]. Перемножение квадратных матриц (MatMult) представлено в 6 вариантах, в зависимости от выбора одного из шести возможных порядков циклов. Триада вычисляет выражение A=B*X+C, где A и B всегда являются массивами, а X и C могут быть как массивами, так и скалярами. В экспериментах было использовано 12 вариантов триады (3 группы по 4 варианта):

1. a(i) = b(i)*x+c
2. a(i) = b(i)*x+c(i)
3. a(i) = b(i)*x(i)+c
5. a(ind1(i)) = b(ind1(i))*x+c
6. a(ind1(i)) = b(ind1(i))*x+c(ind1(i))
7. a(ind1(i)) = b(ind1(i))*x(ind1(i))+c
9. a(ind2(i)) = b(ind2(i))*x+c
10. a(ind2(i)) = b(ind2(i))*x+c(ind2(i))
11. a(ind2(i)) = b(ind2(i))*x(ind2(i))+c

4. a(i) = b(i)*x(i)+c(i) 8. a(ind1(i)) = b(ind1(i))*x(ind1(i)) +c(ind1(i)) 12. a(ind2(i)) = b(ind2(i))*x(ind2(i)) +c(ind2(i))

Во второй группе элементы массива косвенной адресации ind1[i] всегда равны i, т.е. ind1[i]=i, а в третьей группе ind2[i] организован таким образом, чтобы кэш-память использовалась по возможности наихудшим способом.

Для каждой программы было посчитана характеристика Cvg. Значение Cvg, как и локальность, зависит от объема входных данных задачи, поэтому для всех программ использовались входные данные объемом примерно 1 МБ. Небольшой размер входных данных определяется только желанием сократить размер лог-файла, где сохраняется собственно поток обращений, и время на его последующую обработку.

Результаты проведенных нами экспериментов приведены в табл. 1. Значения отсортированы по увеличению характеристики Cvg, при этом, чем меньше Cvg, тем лучше локальность. Абсолютные значения Cvg сейчас не так важны, для нас бóльший интерес представляет ранжирование программ.

Таблица 1. Значения характеристики Cvg для различных программ.

Программа	**Значение Cvg**
MatMult (ikj)	11
MatMult (kij)	11,1
Linpack	12,9
triada 1	17,8
STREAM_1 (a(i) = b(i))	17,8
STREAM_2 (a(i) = q*b(i))	17,8
triada 2	18,6
triada 3	18,6
triada 5	18,6
STREAM_3 (a(i) = b(i) + c(i))	18,6
STREAM_4 (a(i) = b(i) + q*c(i))	18,6
triada 4	19,5
triada 6	19,5
triada 7	19,5
triada 8	20,4
FFT	21
RandomAccess	36,2
MatMult (jik)	38
MatMult (ijk)	38
MatMult (kji)	65,3
MatMult (jki)	65,4
triada 9	91,5
triada 10	100,9
triada 11	100,9
triada 12	106,9

В данной таблице полученные результаты хорошо соответствуют нашим представлениям об эффективности взаимодействия программ с памятью. Например, тест Linpack известен своей хорошей локальностью, что и подтверждается нашими результатами. Далее, 8 вариантов триады и 4 варианта STREAM обладают примерно одинаковой локальностью, и это логично, поскольку во всех этих программах происходит последовательный перебор элементов нескольких массивов, и основная разница только в числе массивов. При этом с точки зрения работы с памятью вариант STREAM_1 и STREAM_3, а также STREAM_2 и STREAM_4 ничем не отличаются, как и соответствующие значения Cvg. Также можно увидеть, что тест RandomAccess обладает достаточно плохой локальностью, однако, что может показаться удивительным, не хуже нескольких вариантов перемножения матриц. Это объясняется следующим. В случае вариантов перемножения матриц jik и ijk элементы одного из массивов перебираются по столбцам, что, с точки зрения локальности, примерно соответствует случайному перебору элементов основного массива в тесте RandomAccess, поэтому эти программы обладают похожей локальностью, что и подтверждают значения Cvg. В случае вариантов kji и jki выполняется перебор по столбцам элементов двух массивов, поэтому локальность этих вариантов заметно

хуже. Самая плохая локальность наблюдается у 4-х последних вариантов триады, и причина этого в том, что в них происходит по возможности наиболее неэффективный перебор элементов сразу нескольких массивов.

Далее можно заметить, что все 6 вариантов для программы перемножения матриц четко разделились по значениям Cvg на 3 группы в зависимости от того, по какому измерению выполняется внутренний цикл. Такая закономерность отлично коррелирует с полученными реальными значениями эффективности, которые приведены в табл. 2. Эффективность определяется как отношение реальной производительности (число операций с плавающей запятой в секунду) к пиковой.

Таблица 2. Корреляция группировки значений Cvg с эффективностью выполнения различных вариантов перемножения матриц.

Вариант перемножения матриц	Значение Cvg	Эффективность, % (Intel Xeon X5650)
ikj	11	13,9
kij	11,1	13,9
jik	38	8,8
ijk	38	9,5
kji	65,3	4,8
jki	65,4	4,8

Важной особенностью характеристики Cvg является ее относительная независимость от свойств выбранного процессора, поскольку в ней учитывается только самое общее представление о строении подсистемы памяти. Именно таким образом мы пытаемся выделить машинно-независимые свойства программ, не привязанные к конкретной вычислительной платформе.

5. Заключение

В данной статье вводится простая характеристика для оценки эффективности работы программ с памятью на основе анализа локальности обращений. Предлагаемая характеристика Cvg позволяет выполнять сравнение свойства локальности обращений в память одних программ относительно других. В дальнейшем мы планируем с помощью данной характеристики или ее модификаций научиться составлять подробное описание локальности для каждой программы, а также предлагать рекомендации по улучшению свойств локальности, и, как следствие, повышению эффективности работы программ с памятью.

Уже сейчас ясно, что в дальнейшем необходимо ввести специальное понятие эффективности для оценки качества работы программ именно с памятью, поскольку используемые сегодня понятия всегда затрагивают и арифметические операции. Новое понятие эффективности работы с памятью необходимо соотнести с введенной характеристикой Cvg.

Не менее важно перейти к рассмотрению реальных приложений из различных предметных областей. При накоплении достаточного объема знаний по реальным приложениям планируется выделить классы задач в зависимости от свойств локальности, а также определить и описать характерные для данных классов потоки обращений.

Работа выполнена при поддержке гранта РФФИ № 13-07-00787 и стипендии Президента Российской Федерации молодым ученым и аспирантам № СП-6815.2013.5.

ЛИТЕРАТУРА:

1. W.A. Wulf and S.A. McKee. “Hitting the Memory Wall: Implications of the Obvious,” Computer Architecture News, vol. 23, no. 1, Mar. 1995, pp. 20–24.
2. Воеводин Вл.В. Суперкомпьютеры и парадоксы неэффективности // Открытые системы. 2009, N10, С17-20.
3. J. Weinberg, M. McCracken, E. Strohmaier, A. Snavely. Quantifying Locality In The Memory Access Patterns of HPC Applications // In Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05), p. 50.
4. Вад В. Воеводин. Визуализация и анализ профиля обращений в память // Вестник Южно-Уральского государственного университета. Серия Математическое моделирование и программирование, т.17(234), 2011, с.76–84.
5. Документация по программному инструменту ThreadSpotter. http://www.roguewave.com/portals/0/products/threadspotter/docs/2011.1/manual/index.html
6. Пользовательская документация по Valgrind. http://valgrind.org/docs/manual/manual.html
7. Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D. The HPC Challenge (HPCC) Benchmark Suite // SC06 Conference Tutorial, IEEE, Tampa, Florida, November 12, 2006.
8. Исходный текст последовательной версии теста Linpack. http://people.sc.fsu.edu/~jburkardt/cpp_src/linpack_bench/linpack_bench_s.cpp
9. Описание вариантов теста STREAM. http://www.cs.virginia.edu/stream/ref.html#counting