Collecting and Exploiting Cache-Reuse Metrics

Josef Weidendorfer[1] and Carsten Trinitis[1]

Technische Universität München, Germany
{weidendo,trinitic}@cs.tum.edu

Abstract. The increasing gap of processor and main memory performance underlines the need for cache-optimizations, especially on memory-intensive applications. Tools which are able to localize code regions with high cache miss ratio seem to be appropriate for access optimizations. However, a programmer often does not know what to do with the collected information. We try to improve this situation by providing cache reuse metrics which are supposed to give more precise hints on how to optimize memory access behavior. We enhanced the cache simulator Callgrind to give metrics on temporal and spatial cache utilization for a given memory block, relating this information to the code line where the block was loaded into cache. We show what is needed for hardware-supported measurement for such metrics, and give example code where the collected information directly points to optimization possibilities.

Keywords Cache Reuse Metrics, Profiling, Cache Simulation.

1 Introduction

A well-known problem for memory-intensive applications is the so-called memory wall: while CPU performance still increases according to Moore's Law at a rate of around 70 % per year, main memory performance improves only by around 7 % per year, leading to an enlarging gap [15]: on modern processors, main memory latency is already around 200-300 processor cycles, causing a large speed impact. This bottleneck will become even more obvious with multi core, multi threaded processors, which quite easily can lead to a fourfold bandwidth requirement.

To reduce this problem, two strategies can be applied: the first is cache optimization by reordering memory accesses or rearranging data-layout, to improve temporal and spatial locality of an application [8]. This approach aims at optimal cache utilization. The second way is prefetching, i.e. pre-loading data into cache which is needed in the future, and thus trying to exploit the available transfer rate to main memory. For both techniques, to locate the code positions where according changes are required, profiling tools are used. These tools relate cache events happening to the code executed at these points in time. Quite simply to implement, and provided in most contemporary processors, are hardware performance counters. Either the application code is instrumented around interesting code regions for gathering exact counter differences, or statistical sampling is used, providing an estimation of the event distribution over the code range. The

latter is done by interrupting the execution every n-th event, and building up a histogram over the program counter at interrupt times.

The described tools provide code ranges where most of the cache misses are happening, i.e. most of the time is spent for memory-intensive applications. However, cache misses are symptoms of memory accesses with bad locality, and not at the same time necessary their cause: e.g. a non-iterative read-stream may pollute the cache and lead to all the misses in the application, but this may happen because other data may have bad access locality: in this case, the tool points to the wrong position. This example may seem to be contrived, and in fact, usually most memory blocks loaded into cache are used more than once, thus being subject to eviction themselves: i.e. the distribution over block-evicting code positions may be similar to that over block-evicted code positions. Still, by relating cache misses to code positions where the evicted lines were loaded, one level of indirection can be avoided. Fig. 1 shows an optimization which gets rid of two misses, and which needs `a` (or the code accessing `a` to be shown as hint for the programmer. Besides, if a cache is enlarged by a small amount, evictions usually happen at a later point in time, and possibly at other code positions, but evicted lines will be the same.

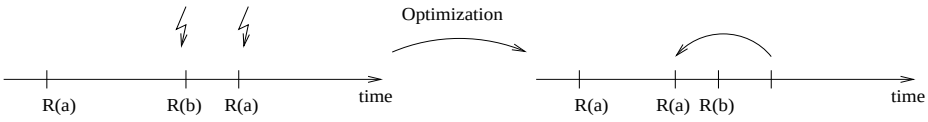

Fig. 1. Optimization by reordering accesses of a (b conflicts with a)

Cache reuse metrics are good to provide more information about cache utilization: the average temporal reuse of a memory block by a cache is given by the number of times the cache was accessed for the memory block divided by the number of times the block was loaded from the next memory hierarchy level. Spatial locality can be measured by the percentage of bytes accessed in a loaded memory block before it is evicted again. Finally, an interesting metric is the stack reuse distance of two sequential accesses to the same memory block: this is the number of memory blocks accessed in-between [3]. This metric, assuming full associativity, is independent of cache size, and gives a precise prediction of when an access will be a cache miss: namely, if the stack reuse distance of this access and the last access to the same location is bigger than the cache size. All these metrics are useful if related to the code positions where the memory block was loaded, and not where the block was evicted (the latter is not applicable to stack reuse distances).

In Chapter 2, we explain the extensions implemented in the cache simulator Callgrind [14], based on the motivation just mentioned. This extension provides metrics for temporal reuse and spatial use of cache lines loaded, related to the code positions where they where loaded. The cache simulator is driven by using runtime instrumentation on the execution of unmodified x86-binaries on Linux

with the help of the Valgrind instrumentation framework [13]. The produced profile data can be visualized without modification in KCachegrind[1]. In Chapter 3, we give some simple examples on the usefulness of the collected metrics for cache optimization. In chapter 4, we present ideas on how to implement the collection of these metrics in hardware, and in Chapter 5, we present some related work. The last chapter gives future research directions.

2 Cache Simulation Additions

Callgrind is an enhanced version of Cachegrind, a cache simulator based on the Valgrind instrumentation framework mentioned above. Only user level code is used to drive the simulation, but this is enough for most applications. In addition, Callgrind builds the call graph of a program on the fly, relates inclusive cost metrics to call chains, and is able to separate cost for different threads. This allows the visualization tool KCachegrind to show exclusive and inclusive metrics for annotated assembler, source, and functions; in addition, it provides an interactive call graph view and a tree-map visualization, showing the inclusive costs graphically in a tree. Extending Callgrind to produce further metrics for source code positions is supported by KCachegrind without any modification.

To allow for cache block utilization metrics and its relation to the code positions that loaded the block, a further data structure was added for every cache block of the L1 and L2. This structure contains the following data for a memory block which currently resides in the corresponding cache:

- the code position of the instruction that recently has filled the block (i.e. the call chain and instruction address),
- a reuse count, and
- a block mask, able to hold the bytes that were touched at least once (a 32bit mask is used, and thus, for 64 byte cache lines, this mask gives a 2-byte granularity).

For every data reference, the runtime instrumentation provides the address and length of the access. These are used in the cache simulation. For an L1 hit, the reuse count and the mask of the according reuse info structure is updated. For an L1 miss, the mask is converted to a byte count, and this and the reuse count are added to counters which are attributed to the stored code position, called "Temporal Reuse L1", and "Spatial Reuse L1". The to-be-evicted block can also be found in the L2. By maintaining according pointers, we can update the reuse metrics for the block in L2: the reuse count is added to the L2 reuse count, and spatial use masks are OR'ed. Afterwards, the L1 reuse counters are initialized for the new data which was loaded on the L1 miss. On an additional L2 miss, the reuse metrics are similarly attributed to counters called "Temporal Reuse L2", and "Spatial Reuse L2".

[1] KCachegrind and Callgrind both are available under the GNU Public License at `http://kcachegrind.sf.net`

It would be quite easy to relate the reuse metrics to code position tuples, adding the code position that evicted the respective cache lines. Currently, this is work in progress on the side of the visualization tool.

3 Results with example code

Cache reuse metrics can pinpoint the following problems:

- code regions with low spatial access locality,
- code regions with low temporal locality.

Low spatial and temporal locality is only important if the number of memory accesses in these regions make up a large portion of all accesses of the application run.

By default, the cache simulator uses cache attributes for L1 and L2 which are the same as on the machine where the simulation is run. In the following examples, we use a Pentium M with 32 KB L1 and 1024 KB L2 cache size; line size is 64 bytes and associativity is 8 for both.

The code example for the first problem iterates over a 2D matrix in the wrong index order, thus emphasizing low locality. Table 1 shows the difference of cache reuse metrics annotated to code using the right (1) and wrong (2) index order. Loads for index variables are not visible because of a initialization loop over the matrix, i.e. aside from read accesses to `matrix[i][j]`, no L1 misses are observed. As a lot of columns fit into L2 cache, the L2 load number for the wrong index order is smaller than L2 loads for the correct order, but reuse for L2 is optimal in all cases. So this example shows only bad behavior regarding L1.

Table 1. Iteration over a 2D matrix

	Code	L1 Loads	L1 Spat. Reuse	L1 Temp. Reuse	L2 Loads	L2 Spat. Reuse	L2 Temp. Reuse
	`int matrix[1000][1000]`						
	`...`						
	`for(i=0;i<1000;i++)`	0			0		
	`  for(j=0;j<1000;j++)`	0			0		
(1)	`    sum += matrix[i][j]`	62 500	100 %	16	62 500	100 %	16
(2)	`    sum += matrix[j][i]`	999 992	6 %	1	60 190	100 %	16

The example for low temporal locality does a matrix-matrix multiplication without blocking. Table 2 similarly shows the difference of cache reuse metrics annotated to the unblocked (1) and the blocked (2) version of this code. L2 reuse metrics show a quite good behavior because of the quite small matrix size.

For larger programs, first, one has to locate the code regions with bad locality before being able to look at details like the ones outlined above. One has to find

Table 2. Matrix multiplication unblocked and blocked

Code	L1 Loads	L1 Spat./Temp. Reuse	L2 Loads	L2 Spat./Temp. Reuse
(1) `for(i=0;i<1000;i++)`	0			
`for(j=0;j<1000;j++)`	0			
`a[i][j] += b[i][j] *` `b[j][i];`	1 123 993	68 % /2.65	176 398	94 % /15
(2) `for(ii=0;ii<1000;ii+=50)`	0			
`for(jj=0;jj<1000;jj+=50)`	0			
`for(i=0;i<50;i++)`	0			
`for(j=0;j<50;j++)`	0			
`a[i+ii][j+jj] +=` `b[i+ii][j+jj] *` `b[j+jj][i+ii];`	219 513	81 % / 13.0	175 791	100 % / 16

a derived event type formula using miss counts and reuse metrics, such that sorting the resulting costs over the whole program allows to spot the important bottlenecks easily. To spot code lines with low temporal locality, a ordering of the lines according to the formula

$$MissCount/TemporalReuse$$

should be used, and for low spatial locality

$$MissCount * (1 - SpatialReuse)\,,$$

using the metrics attributed to one line each.

4 Hardware Support

To build caches which are able to monitor cache reuse, the additionally used circuit space should to be minimized. An extension which produces data on every cache miss would almost need a second processor to handle the amount information collected. Thus, counters which are used in a statistical sampling method seem adequate.

We want to relate the reuse metrics to the code position which triggered the load of that cache line. The simplest solution would be to store this code position for every cache line on a cache load. Unfortunately, there is no easy way to get at this address from the cache controller's point of view, in the presence of out-of-order processors. In addition, this would need quite a large additional amount a chip space. But the cache controller could store the old tag on a cache miss, enabling us to obtain the address of the evicted memory block in an interrupt handler which is triggered at overflow time of a hardware performance counter for cache misses. Looking at Intel P4 and Itanium processors, there is

a possibility to get at the data address which leads to the cache miss (PEBS on P4), or to a long cycle stall because of memory access (special register on Itanium, with "EAR" events). This way, our cache reuse metric at least could be attributed to (evicted,evicting)-data address tuples.

However, relation to source code would be desirable. By simultaneously sampling misses related to (code position, evicting address), we get some meaning of which data structures are accessed at which code positions. By presenting this information to the user in a meaningful way, he or she should be able to make use of it.

For the reuse metrics themselves, some bits have to be spent for each cache block: for spatial locality, a granularity at byte level is not necessarily needed: A bit mask with one bit for a floating point value should be sufficient, resulting in 8 bits per 64 byte cache line. For temporal locality, it should be sufficient to use a saturating counter with e.g. 4 bits per cache line, as a reuse of more than 16 probably does not need to be optimized. However, this is arguable and subject to further research. The cache controller has to be improved for updating these masks and counters, and on a miss, a performance counter has to be incremented by the according amount. Presuming that only the additional bits per cache line influence the additional need chip space, 2.3 % of additional space are needed (12 bits per 64*8 bits).

While it is unlikely that a processor vendor would add these bits for reuse metrics on every consumer processor, allowing the reading of the old tag on a cache miss for an interrupt routine seems plausible.

5 Related Work

Most profilers relate timer or performance counter events like cache misses to code positions, like DCPI [1] (for Alpha), VTune [7] (for x86) or OProfile [9] (Linux), most often using statistical profiling. While this can be done without recompiling, instrumentation is needed to get more information. This way, tools can get exact event counts of code regions and call graphs like Adaptor [4] or VTune. Using cache simulation, runtime instrumentation is most comfortable to get cache events, like in Cachegrind [12], which the simulator in this paper is based upon.

Cache simulation currently is the only way to obtain cache utilization metrics. Memspy [10], SIGMA [5] SIP [2] and MHSIM [11] are examples of cache simulators. The latter two are similar to our work; MHSIM is based on source instrumentation, ignoring compiler optimizations. SIP only calculates metrics at one cache level. It runs on SPARC, simulates a whole system, and produces fixed HTML files. Our tool only applies to user level, but its cache simulation can be switched on dynamically only in interesting program phases, and it is as easy to use as `strace` or `perfex`. It takes advantage of the visualization possibilities of KCachegrind.

Regarding hardware support for enhanced performance analysis, Intel's latest processors seem to provide most features. The number of available performance

counters (18) and types of events measurable, in addition to advanced filter and cascading possibilities, together with PEBS (see Chapter 4), make the Pentium 4 platform a good target for any performance analysis (tools to be written) [6]. The Itanium 2 platform at least has a similar level of support for performance analysis tools, as it is able to filter events on opcode, instruction and data ranges, and provides data addresses for data accesses lasting longer than a given threshold. This abstracts from any cache miss events, and focuses on actual latency, which is regarded as valuable.

6 Summary and Future Work

In this paper, we added support for collection of cache reuse metrics to Callgrind, part of our tool suite for cache analysis. The visualization side, KCachegrind, did not need to be changed for this addition.

Work is carried out on being able to visualize metrics related to tuples of code positions. This will allow us to relate reuse metrics to (evicted, evicting) code positions, and enable for addition of the stack reuse distance metric in the tool. Stack reuse distance usually involves $O(\log n)$ complexity for updating a stack of size n for each access, but a constant time algorithm is available for approximated results (buckets of distance ranges), which should be adequate.

In joint work with hardware people in our group, we want to check the usefulness and complications of adding hardware support for reuse metrics as given in Chapter 4, by modifying a VHDL description of a simple MIPS core, and running it in an FPGA.

Acknowledgments to Julian Seward and Nick Nethercote for their useful and "simply working" Valgrind tools.

References

1. J. M. Anderson, L. M. Berc, J. Dean, et al. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.
2. E. Berg and E. Hagersten. SIP: Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.
3. K. Beyls and E.H. D'Hollander. Platform-Independent Cache Optimization by Pinpointing Low-Locality Reuse. In *Proceedings of International Conference on Computational Science*, volume 3, pages 463–470, June 2004.
4. T. Brandes. Adaptor. Homepage. http://www.scai.fraunhofer.de/291.0.html.
5. L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC 2002*, Baltimore, MD, November 2002.
6. Intel Corporation. IA-32 Intel Architecture: Software Developers Manual.
7. Intel Corporation. Intel VTune Performance Analyser. available at http://www.intel.com/software/products/vtune/.

8. M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, March 2003.
9. J. Levon. OProfile, a system-wide profiler for Linux systems. Homepage: http://oprofile.sourceforge.net.
10. M. Martonosi, A. Gupta, and T. E. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.
11. 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.
12. N. Nethercote and A. Mycroft. The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, July 2002.
13. N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003. Available at http://developer.kde.org/~sewardj.
14. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proceedings of International Conference on Computational Science*, volume 3, pages 455–462, June 2004.
15. Wm. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.