Visualizing the Impact of the Cache on Program Execution

Yijun Yu and Kristof Beyls and Erik H. D'Hollander

University of Ghent
Department of Electronics and Information Systems
St-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{Yijun.Yu,Kristof.Beyls,Erik.DHollander}@elis.rug.ac.be

Abstract

The global cache misses ratio of a program does not reveal the time distribution of the memory reference patterns in detail. On the other hand, cache visualization is hampered by the huge amount of memory references to display. Therefore, many visualizers focus on a snapshot of the cache content, instead of viewing all memory transactions. In this paper, a cache visualizer is introduced which presents the integral cache behavior of a program in several complementary views. The density view of the cache misses shows the hot spots of the program; the reuse distances view shows the data locality and its effect on performance; the histogram view shows the periodical patterns that occurs in the trace. In a number of experiments, the visualizer has been used to characterize the cache behavior and effectively improve the cache behavior and program performance.
Keywords *program visualization, cache, reuse distance, data locality, loop tiling*

1. Introduction

Modern architectures face an ever-widening gap between the memory speed and the processor speed. Using a cache, a small but fast memory, the delay of a memory access is reduced when the requested address is already stored in the cache. Otherwise, a cache miss occurs, which stalls the processor until the memory access is done. With a memory access time exceeding 10-20 times the delay of a cache hit, cache misses become all important.

The impact of the cache on the performance of a program is measured by the cache miss ratio, i.e. the fraction of all memory accesses not in the cache. This ratio is determined by the memory trace, the sequence of memory accesses of a program in execution. It is also influenced by the cache hardware parameters, such as the cache size, cache line size, set associativity and replacement policy.

There are many tools to measure the cache miss ratio, either through cache simulation [10], such as Dinero [6], Cprof [8] or sampling from hardware counters like VTune [7, 1]. There are also compiler techniques to estimate the cache miss ratio analytically [4].

In order minimize cache misses, transformations to improve the data locality [9, 12], such as loop tiling, loop fusion, array padding and array alignment, have been proposed. However, in order to understand the cache behavior in general programs, one needs to look at the hit rate during the execution. Several tools have been introduced to visualize the cache behavior. For example, CVT [11] provides a visualization of the cache lines during the program simulation. Cache parameters are allowed to be reconfigured and the effect can be stepwise observed.

Rivet [2] visualizes cache behavior through statistical histograms of the cache lines. The histograms show which cache lines are more frequently used.

Both CVT and Rivet visualize the distribution of references along the cache lines, because the number of cache lines are relatively small in comparison to the number of memory references of a program. However, this doesn't allow to visualize the cache performance of a whole program, because the cache content is frequently refreshed and the huge data space of a program is observed trough the tiny cache window. This makes it difficult to recognize the data access patterns generated by the program.

Therefore, an alternative way is presented in this paper, which is program rather than cache centered. The millions of memory references are shown in a compact view that reveals the global temporal patterns of the program execution and their impact on the cache behavior.

In the following section the cache model and performance measures are presented. In section 3, different visualization views are developed which help the programmer to recognize cache access patterns and give hints to improve them. In section 4, the cache visualizer is applied to a number of programs, and it is shown how it is possible to

increases the hit rate and the program execution. The observation of the changed density and distribution gives more information on why the number of cache misses is reduced by a program transformation.

2. Cache representation

A *cache line* is the block of aligned consecutive bytes each load or store operates on. Define the size of the cache as Cs bytes and the size of a cache line as Ls bytes. Similar to a cache line, the block of consecutive bytes moved between the memory and the cache in one transaction is called a *memory line*. An access to the memory address can *hit* the cache if its memory line is found in the cache, or *miss* the cache if its memory line is not found.

Each memory line can be placed in K different lines of the cache. K is called the associativity and a cache is called *K-way associative*. In particular, if $K = Cs/Ls$, the cache is called *fully associative* and if $K = 1$, it is called *direct-mapped*. The set of K cache lines that a memory line can be mapped onto is called a *cache set*. For a fully associative cache (FAC), the only cache set is the whole cache. For a direct-mapped cache (DMC), each cache line is a distinct cache set.

The cache misses are distributed in time and categorized as compulsory misses, capacity misses and conflict misses [6]. Compulsory (cold) misses occur the first time a memory address is cached. Conflict misses occur when a cache set has to make room for a new memory access, while there is still room in the cache. Capacity misses are generated when the cache is full and a new memory line enters the cache.

2.1. Reuse distance

An important program parameter to visualize is the reuse distance. There is a *reuse* if the same memory line is used again in the program. The *reuse distance* is defined as the number of distinct memory lines fetched between two accesses of the same memory line. It is also called the *stack distance*.

Reuse distance is a good measure to indicate the cache misses in a fully associative cache (FAC). In a FAC, a miss happens either because it is the first time to access the memory line, or it is a reuse of a memory line with reuse distance greater than the number of cache lines Cs/Ls. The former case is called a *compulsory* or *cold* miss, the latter case is called a *capacity* miss. Cold miss obviously can not be avoided, but capacity misses may be removed by increasing the cache size.

If a miss in a K-way associative cache would not happen in a fully associative cache of the same size, it is a *conflict* miss. To indicate a conflict miss, the reuse distance can be applied to a cache set: the *set reuse distance* is the number of distinct memory lines that are fetched in the same cache set between two accesses of the same memory line. We call the reuse distance in a fully associative cache the *FAC reuse distance*.

When a reuse distance is greater than K, and the corresponding FAC reuse distance is also greater than Cs/Ls, this reference is a capacity miss; if the corresponding FAC reuse distance is less than or equal to Cs/Ls, this is a conflict miss.

In order to reduce the number of cache misses, therefore, it is often desirable to minimize the reuse distances.

2.2. Program instrumentation

To visualize the dynamic cache behavior of a program, it is natural to show a trace-driven simulation of the memory access patterns.

A trace-driven simulation requires to instrument the program in order to get an address trace of memory loads and stores. This is done either by inserting library function calls or output statements at each data reference.

Instead of instrumenting the binary program, we instrument the source program in a compiler to get the same result. The drawback of source code instrumentation is that its memory use is not exactly the same as that of an optimized program; on the other hand, the advantage of source code instrumentation is the possibility to trace back the exact line in the source program.

Another limitation of trace-driven simulation is that the logged trace data is huge in a memory-intensive program. Compression techniques allow it to be shrunk to one-ninth of the plain text. Still a rapid access is required in order to efficiently visualize the dynamic cache behavior of the the touched memory lines. This is solved using a balanced AVL tree data storage which allows data access in $O(\log N)$ time, where N is the number of used distinct memory lines.

3. Visualization

The cache behavior of the program trace is visualized in a 2D frame. In order to present the cache behavior of the millions of memory references in the whole program efficiently, a memory access is represented by a pixel, coded by a color. Consecutive memory accesses are represented by adjacent pixels, and the pixel lines are horizontally wrapped. In this way, the program behavior becomes immediately visible as a pattern. The global pattern allows to visualize the cache behavior of a program in one single view, and the superb pattern recognition capabilities of man are used to discern different cache behaviors. Examples are a regular data access image or hot spots indicating poor cache behavior.

Several possible filters are available to reveal different aspects of the program cache behavior.

3.1. Density of cache misses

The distribution of the cache misses during program execution is necessary to identify the areas of congestion. In this view, cache hits are white and cache misses are colored. In order to classify the cache misses, each type is painted in a different color (or grey scale): blue (black) for a compulsory miss, green (dark grey) for a capacity miss and red (light grey) for a conflict miss. The detailed distribution of the colored pixels and their density is more useful than just having the total number of misses. An example is given in figure 2. It shows the regular access patterns of a matrix multiplication, with about 30% cache misses, mostly capacity misses. This suggests that the cache is not optimally used.

3.2. Reuse distances

A second view shows the reuse distances. Each reference is colored with a value according to reuse distance defined in section 2.1. One can select between two types of reuse: set and fully associative. Using the set reuse distance you can see the pattern of conflict misses, while the fully associative reuse distances show the capacity misses. An example of this view is shown in figure 8, again for the simple matrix multiply.

3.3. Histogram of the grouped references

A histogram view is used to analyze the regularity of recurring reference patterns or reuse distances. The view represents horizontally the patterns to analyze (e.g. reuse distances), and vertically the number of identical patterns. For example, the figure 9 shows the histogram of reuse distances with only three larger peaks, indicating that the distance between consecutive memory lines is very regular.

Finally, in the cache visualizer it is possible to highlight a group of references to the same memory line or to an array element by left or right clicking on an individual reference in the trace view. Then a histogram can be built based on the selected references. This allows to relate the cache view to the data layout of the program.

4. Experiments

In the following experiments, the density and distribution patterns of cache misses, the hot spots of reuses, the histogram of the reuse distances are shown for several programs. The comparison of the cache behavior between transformed program and the original program indicates whether the transformation is effective. More details can also be found in our website [3].

```
#define N 40
double a[N][N];
double b[N][N];
double c[N][N];
void mxm() {
  int i,j,k;
#pragma do_trace
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      c[i][j] = 0;
      for (k=0;k<N;k++) {
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
      }
    }
  }
}
```

Figure 1. The matrix multiplication example. The pragma statement indicates to generate a memory reference trace of the arrays. An instrumentation compiler based on SUIF is used to generate the trace output automatically.

4.1. Visualizing the density of cache misses

As a simple example, consider a matrix multiplication program in figure 1.

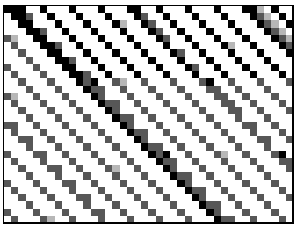
The pragma statement indicates that the compiler should instrument the program to generate a memory reference trace. The instrumentation is done using the SUIF compiler [5]. The cache simulation shows a configuration of 1 KB direct-mapped cache with 32 byte cache line size. The cache miss patterns are shown in figure 2.

The cache miss ratio is 30%. Cold misses are shown in blue, mostly in the upper left. As can be expected, the beginning of the program suffers cold misses more frequently to arrays A and B. However, due to the intermittent access to matrix C, a few cold misses happen occasionally till the end of the trace (slanted downward line in the figure).

The capacity misses are shown densely in green. This pattern comprises most of the cache misses. The large number of capacity misses may indicate that the cache size is relatively small. The pattern looks rather regular, since matrix B is thrown out of the cache at each iteration of the i loop.

The conflict misses are shown in red, evenly spread over the whole trace in a regular pattern. Conflict misses are rather low compared to the capacity misses.

(a) The whole program



(b) Upper left corner 40x30 references enlarged

Figure 2. Cache miss patterns of the matrix multiplication.Black (blue) = cold, dark grey (green) = capacity, light grey (red) = conflict misses.

Table 1. Cache misses reduced by tiling

number of	matmult	ratio	tiled	ratio
refers	257600	100.0%	257600	100%
cold	1200	0.5%	1200	0.5%
capa	72366	28.0%	10730	4.2%
conf	5340	2.1%	16900	6.5%
misses	78906	30.6%	28830	11.2%

4.2. Visualizing the improvement of tiling

A better cache behavior can be obtained when the multiplication is reorganized as figure 3 using loop tiling. Now the inner loops perform the calculations, such that the resulting ”tile” of matrix C is completely obtained from a single read of the corresponding tile in matrices A and B. The outer loops make sure that every tile in matrix C is calculated. The effect on cache behavior is visible in figure 4. and the improvement over the untiled version is shown in table 1.

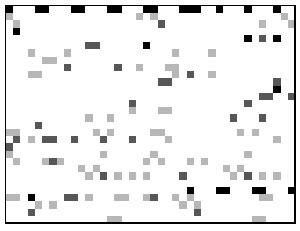
4.3. Visualizing effects of cache organization

The size of the cache affects the total number of capacity misses and the set-associativity affects the number of con-

```
#define min(a,b) ((a)<(b)?(a):(b))
#define B1 5
#define B2 5
#define B3 5
void mxm_tiled() {
  int i,j,k;
  int I,J,K;
#pragma doisv
  for (I=0;I<N;I+=B1)
    for (J=0;J<N;J+=B2) {
      for (i=I;i<min(I+B1,N);i++)
        for (j=J;j<min(J+B2,N);j++)
          c[i][j] = 0;
      for (K=0;K<N;K+=B3)
       for (i=I;i<min(I+B1,N);i++)
        for (j=J;j<min(J+B2,N);j++)
         for (k=K;k<min(K+B3,N);k++)
           c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
}
```

Figure 3. The tiled matrix multiplication. The inner loops perform the calculations, such that the resultant "tile" of matrix C is completely obtained from a single read of the corresponding tile in matrices A and B. The outer loops make sure that every tile in matrix C is calculated.

(a) The whole program



(b) Upper left corner 40x30 references enlarged

Figure 4. Cache miss patterns of the program in figure 3.

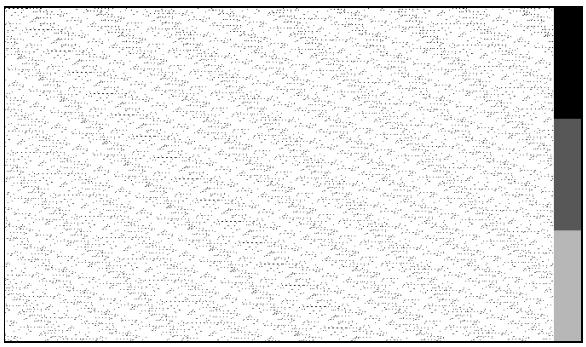


Figure 5. The view of cache miss patterns of tiled version with a 1kb FAC.

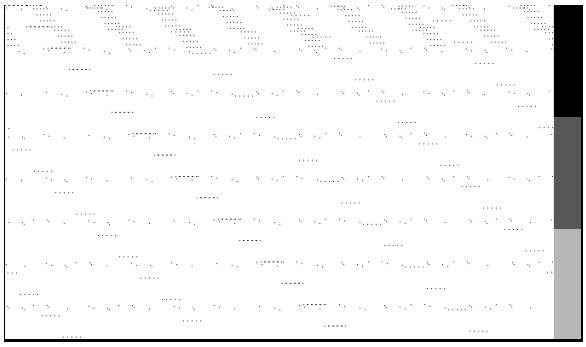


Figure 6. The view of cache miss patterns of tiled version with a 32kb FAC.

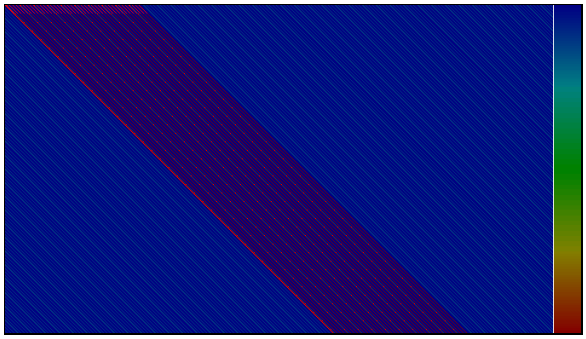
flict misses.

For example, the memory trace of of a 1K fully associative cache is shown in figure 5. There are no conflict misses (red color). This means the conflict misses are removed in a 1K fully associative cache. Enlarging the cache to 32K shows only cold misses (blue), which means both the capacity and conflict misses are removed (figure 6).

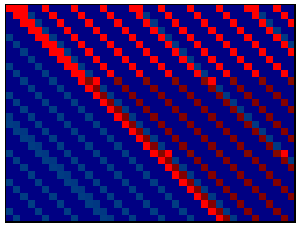
4.4. Visualizing the reuse distances

In figure 7, the fully associative cache (FAC) reuse distances of the untiled program is shown. The value of the reuse distance is indicated by a color, from blue to red, for values from small to large. This allows to trace the hot spot patterns in the memory trace. One can see that the hot spots are closely correlated with the patterns of cold misses and capacity misses in figure 2.

In order to reduce capacity misses, a tiling loop transformation is applied, as shown in figure 3. The FAC reuse distance of the execution is shown in figure 8. One can see that there are much less “hot” reuse distances after tiling compared with figure 7.

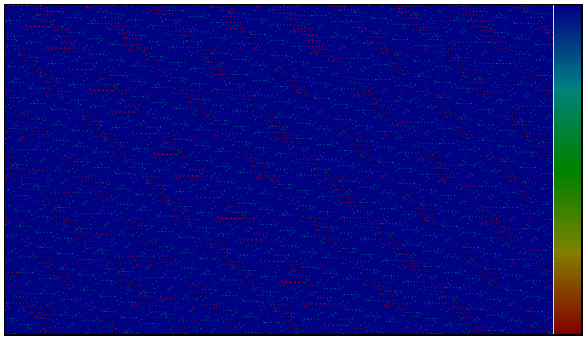


(a) The whole program

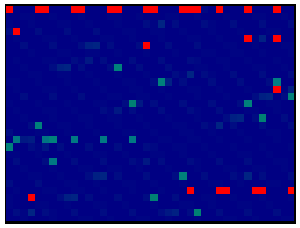


(b) Upper left corner 40x30 references enlarged.

Figure 7. The view of FAC reuse distances of matrix multiplication with a 1KB FAC.



(a) The whole program



(b) Upper left corner 40x30 references enlarged.

Figure 8. The view of FAC reuse distances of tiled multiplication with a 1KB fully associative cache.

4.5. Histogram of reuse distance

In figures 9 and 10, the histograms of the reuse distances are shown for two the original program and the tiled version. The peaks of the reuse distances indicate that the memory references are rather periodical. This is also visible in the reuse distance view shown in figure 7. Furthermore, in the tiled version fewer reuse distances exceed the number of cache lines than in the original program, which is the goal of tiling. As a consequence, the data remains longer in the cache, which improves the program performance.

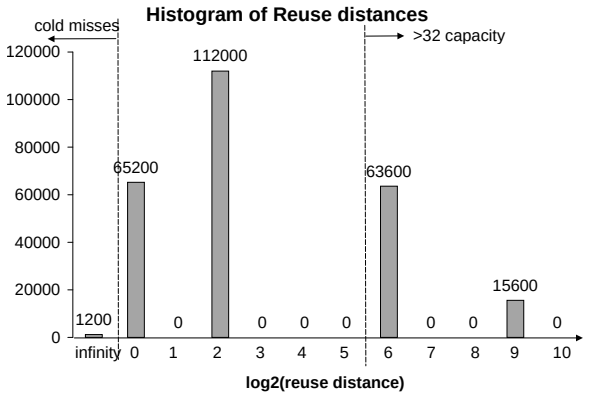


Figure 9. The histogram of FAC reuse distances for matrix multiplication with a 1KB fully associative cache. The cold misses are represented as "infinity" distances and the distances greater than the number of cache lines (32) lead to 79200 capacity misses.

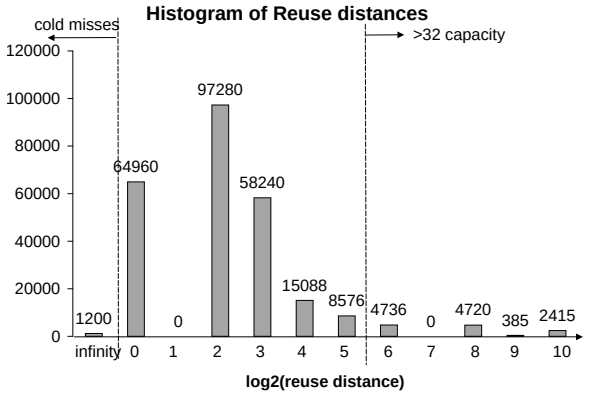


Figure 10. The histogram view of FAC reuse distances for tiled multiplication with a 1KB fully associative cache. The total of distance greater than 32 decreases to 13456. Therefore the capacity misses are 17% of figure 9.

5. Conclusion

We have shown that huge amounts of cache misses and hits can be adequately represented in a small image-like pattern, giving valuable information to the programmer. This information can be used by the programmer to improve the execution time using a better data layout or change the instruction order using transformations such as tiling. In some instances, the visualizer can guide the programmer in writing a more cache efficient algorithm. It is believed that further experience with the pattern cache visualizer will show it to be a valuable tool to overcome the growing processor-memory distance.

References

[1] M. Atkins and R. Subramaniam. Pc software performance tuning. *Computer*, 29(8):47, Aug 1996.

[2] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34(1):68–73, Feb. 2000.

[3] E. D'Hollander, Y. Yu, and K. Beyls. Parallel programming tools. *http://elis.rug.ac.be/paris/ppt.*

[4] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *ACM SIGPLAN Notices*, 33(11):228–239, NOV 1998.

[5] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84, Dec. 1996.

[6] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[7] Intel. Intel VTune performance analyzer. Technical report, Intel, Corp., 2001.

[8] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks - a case-study. *COMPUTER*, 27(10):15–, Oct. 1994.

[9] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, JUL 1996.

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

[11] E. vanderDeijl, O. Temam, E. Granston, and G. Kanbier. The Cache Visulization Tool. *IEEE Computer*, 30(7):71, July 1997.

[12] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, Oct. 91.