CCAI Vol. 19, 3 - 4

VISUALIZING THE IMPACT OF THE CACHE ON PROGRAM EXECUTION

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

Abstract

Cache behavior of a program has an ever-growing strong impact on its execution time. Characterizing the behavior by visible patterns is considered a way to pinpoint the bottleneck against performance.

This paper presents a framework of visualization for trace distributions to extract the useful cache behavior patterns. We focus on cache misses, reuse distances, temporal or spatial localities, etc. The histograms of these distribution patterns measure the behavior in quantity, revealing effective program optimizations. The performance bottlenecks are exposed as hot spots highlighted in the source code, showing the exact locations to apply suitable optimizations. The impact of the source-level program optimizations, again, can be verified by the visualization tool.

Keywords

Program visualization, cache, reuse distance, data locality, performance optimization

1 Introduction

Modern computer architectures face an ever-widening gap between the memory speed and the processor speed. Using a cache to compensate the delay of a memory access is a common practice in architecture design. With a memory access time exceeding 10-20 times the delay of a cache hit, cache hit ratio, i.e. the fraction of those memory accesses hit the cache, becomes all important.

Tuning the cache hit ratio involves both software and hardware issues. By the software, the sequence of memory accesses of the executing program, also called address trace, determines the program's behavior during execution; by the hardware, the cache parameters such as the cache size, cache line size, set associativity and replacement policy strongly influence the program's cache behavior.

Accordingly, there are three ways to improve the cache performance:

1. Equip with better hardware. In principle, the largest cache size and cache line size, fully set associative cache with the optimal replacement policy are pursued. However, limited by the chip resources, it becomes too expensive or infeasible to employ.

2. Complete analysis in compiler. With an all-capable compiler, a -O4 option solves everything. It does the data layout arrangement and statements reordering in an optimal way such that the cache misses are fully compensated by the processing of useful computations. This is yet, however, an ideal that no existing compiler can fulfill. The reason for that is the limited analytical power and limited intelligence to predict.

3. Visualize the bottlenecks. A doctor can not find bacteria without a microscope, so does a performance optimizer. He or she has to imagine what is going on in the invisible chips and the logical relation between cache and the program execution. With the assistance of the cache visualization tools, a programmer will no longer be blind to cache behavior of the program execution.

In this paper, a framework of the third approach has been illustrated with the aim to cache performance optimization. Experiencing the use of it, an alternative solution to the hard problem can be found.

In the following section models are briefly explained for classifying the cache misses, measuring reuse distances and data localities. In section 3, the visualization framework is illustrated with the development and use of three views: 1) the trace view captures memory access patterns in time; 2) the histogram view measures patterns in quantity; 3) the hot-spot highlighting view reveals the bottlenecks in source code. In section 4, the cache visualizer is applied to several programs. The visualization of the optimized program observes the effects of cache hits improvement. In section 5, the method is related to the others.

2 Cache Models

A *cache line* is the block of aligned consecutive bytes each load or store operates on. Define the size of the cache as *cache size* Cs bytes and the size of a cache line as *line size* 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 either *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.

2.1 The 3C misses

The cache misses are categorized as compulsory misses, capacity misses and conflict misses, called the *3C misses* [hill89]. *Compulsory* (also called 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.

Basically, compulsory misses are related to the size of a program's work set, capacity misses are related to the limiting size of the cache, conflict misses are related to the limiting associativity of the cache.

2.2 Reuse distance

An important parameter to indicate the data locality of a program 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* by Belady [belady66].

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 the reuse distance greater than the number of cache lines *Cs/Ls*. The former case is called a compulsory or a 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 measurement 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. To distinguish from the case of fully associative cache, we call the reuse distance in a fully associative cache a *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.

2.3 Temporal vs. spatial locality

According to McKinley and Temam [mckinley99], a *locality* is the reuse distance between two references R_1 and R_2 to the same memory line. A locality is *temporal* when the address of R_1 and R_2 are equal; a locality is *spatial* as the address of R_1 and R_2 are different and R_1 misses the cache.

It is useful to classify the reuses as temporal or spatial locality because 1) the locality is the feature of the program independent of the cache size; 2) the techniques to exploit temporal locality are different from those for spatial locality.

Temporal locality optimizations such as loop tiling and fusion are *control* oriented, while spatial locality optimizations like array padding, alignment and packing are *data* oriented. Thus visualizing the locality helps programmer to select suitable optimizations.

3 Approaches

The visualization of program's cache behavior presents the program with its execution traces, histograms and hot shots. In addition, these individual views must be related to present a methodology that can be applied to study practical programs.

The skeleton of cache behavior investigation is explained as follows:

1. Instrumentation: A source program is subjected to an automatic instrumentation profiler for coding the address trace during the program execution.

2. Visualization: Visualizations are applied to the address traces for programmer to identify the location of the bottleneck among 1) Poor data alignment due to array locations; 2) Conflicted cache lines due to loop strides; and 3) Small cache capacities due to huge address space or loop work set. The visual patterns reveals the dominate bottleneck that influenced the performance.

3. Optimization: Based on the visualized bottleneck, a proper optimizing technique is employed to 1) Reduce the conflict miss and improve spatial locality by careful address calculation: data alignment, padding, and increase set-associativity; 2) Reduce compulsory miss by compaction of data set e.g., array packing; and 3) Reduce capacity miss and improve temporal locality by compaction of work set e.g., loop fusion and loop blocking.

4. Verification: The effects of program transformation can be visualized to see the resolved dominate patterns. To see the new dominate patterns for further optimization, go back to step 1.

There are two subsystems, instrumentation and visualization in the cache visualizing system, as shown in Figure 1.

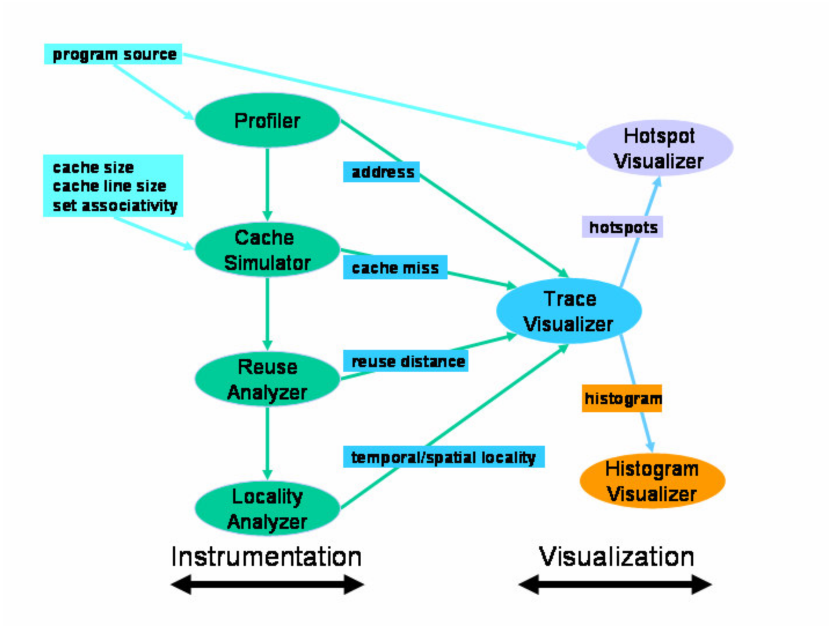


Figure 1. The cache visualizing system

Instrumentation: A source program is instrumented by the *profiler*. The execution of the instrumented program generates the memory address traces of the array references in the program, additional information like array names, reference numbers and program unit numbers (loops or loop nests), etc. are also output as traces.

The address trace, together with the cache hardware parameters, are used by the *cache simulator* to calculate the traces of cache misses, reused references and cache line numbers.

The cache line number and reused reference traces are further exploited by the *reuse analyzer* to obtain reuse distances and set reuse distances. The backward reuse distances can be used to analyze temporal locality of the program because they are independent of cache size. The forward reuse distances can be used to choose proper cache hints for pre-fetch techniques.

The *locality analyzer* further classifies the reuse distances as either temporal or spatial localities through analysis of both the reuse distance traces and the cache miss traces. A reuse is temporal locality when the two references to the same memory line access the same address; a reuse is spatial locality only when the first reference to the same cache line misses the cache and the latter accesses to its neighboring address.

Visualization: To consume the information produced by the above tools, the visualizers provide the information to the programmer graphically.

The *trace visualizer* presents the time distribution of the traces to reveal periodical patterns. It also generates histograms to measure the patterns in quantity. Any two distribution patterns of the same program can be related and revealed as correlation histograms.

The *histogram visualizer* displays the quantities of the related patterns in bar-charts and sorts out the top 20 most important patterns according to the quantity or the distribution.

Besides the statistical results in histograms, the trace visualizer also distributes the quantities in relation to the source program as hot spots. Being highlighted, it's very convenient for the programmer to find out them outstands in the *hot spots visualizer*.

The detail design of the implementation and visualization are discussed as follows.

3.1 Profiler

To visualize the dynamic cache behavior of a program, it is natural to show a trace-driven simulation of the memory access patterns. Some design choices are made to fulfill this possibility.

A trace-driven simulation requires a *profiler* 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 in the proper order.

The source program instead of the binary program is instrumented in a compiler. One may argue that a drawback of source code instrumentation is that its memory use is not exactly the same as that of an optimized program.

Since compiler writer can output the optimized code in source form, most high-level optimizations can still be compared. Meanwhile, the advantage of source code instrumentation is the possibility to trace back the exact location in the source program.

Because most of the scalar references in the optimized code will be replaced by registers in the optimized program, the remaining array references have the most significant impact on the cache behavior. We choose to instrument array references only, though it's trivial to consider scalar as a one element array.

While the trace data are huge in a memory-intensive program, a rapid access is required in order to efficiently visualize the dynamic cache behavior of 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.

3.2 Visualizers

The *trace visualizer* presents the time distribution of the traces to reveal periodical patterns; the *histogram visualizer* displays the quantities of the related patterns and the *hot spots visualizer* reveals the bottlenecks in source code. They are all explained in this section as follows.

3.2.3 Trace view

The cache behavior of the program trace is visualized in a 2D frame. In order to present millions of memory references in the whole program efficiently, a memory access is represented by a pixel with its value coded by a color.

Each trace view presents one feature of the trace, such as *cache miss*, e.g. figure 2; *reuse distance*, e.g. figure 3.2; *loop iteration*, e.g. figure 4.3; *reference number*; and *temporal/spatial locality*; etc.

Color allocation: As a feature like reuse distance has comparable values, the color codes the value according to the HSB or the grey-scale spectrum, the smallest value starts from blue or white, the largest value ends with red or black, depends on the choice of the HSB or grey-scale spectrum.

As a feature like cache miss has non-integer values, the color coding is configurable. The default assignment is blue for compulsory, green for capacity and red for conflict.

Zooming feature: When the programmer is interested in a region in particular, it has to be zoomed in, when the programmer is interested in the time distribution pattern of the whole program, the trace has to be zoomed out.

It is relative trivial to zoom in or enlarge each pixel into a line segment or a block of pixels. It is not that trivial, however, to zoom out several pixels into one single pixel without loss of pattern of the time distribution of the whole program.

One way is to summarize the RGB components colors of the pixels in average. This will not affects a global distribution pattern; however, there will be less sharp colors. One extreme, after many folds of zoom out, is to have one average color for the whole trace. To avoid this, another way is to summarize the percentage of occurring color within the neighboring region. This will shifts the distribution pattern within the zoom out scale while keeping the sharpness of colors.

Both alternatives are provided in the trace visualization.

Periodical Patterns: Consecutive memory accesses are represented by adjacent pixels, and the pixel lines are horizontally wrapped. Since most programs contain loops that repetitively access memory, many aspects of the trace, like cache misses, reuse distances, have a periodical property. When the period divides the number of pixels in a horizontal line, the periodical patterns can be visually recognized. In this way, the program behavior becomes immediately visible as a pattern. Since the window width is resizable, a pattern with any period length is recognizable.

The global pattern allows visualizing 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.

Patterns in misses: The cache miss trace classify the references into four categories: hit, compulsory miss, capacity miss or conflict miss. The default color coding them in white, blue(light grey), green (grey) and red (dark grey) respectively in HSB (grey scale) spectrum.

The distribution of the cache misses during program execution is necessary to identify the areas of congestion. A detailed distribution of the colored pixels and their density is more useful than just having the total number of misses.

There are basically three patterns to reveal the dominant cache misses. Figure 4 gives some simple examples of these patterns.

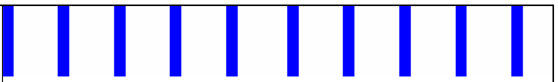
1. Compulsory misses + spatial locality: A cache line holds \$N\$ elements. Thus a compulsory miss brings the neighboring elements into the cache, allowing hits in the N-1 sequential accesses. In Figure 2, the first loop follows this pattern. In this case, array packing can reduce the compulsory misses using smaller array element size to increase N. Increasing the cache line size would have similar effect.

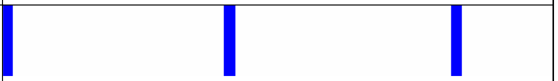
2. Compulsory misses + temporal locality: When the elements are repeatedly reused in a loop, the reference number increases while the compulsory misses does not. Thus the miss ratio is reduced. The second loop nest in Figure 2 has the pattern.

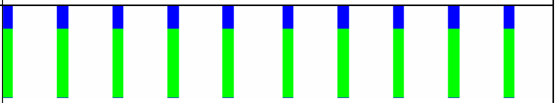
3. Capacity misses due to array size >> cache size: Many capacity misses when the cache size is exceeded before an element of a larger array can reuse it. The third loop nest just interchanges the two loops of the second loop nest, resulting in this pattern. Possible solutions are loop blocking and fusion.

4. Conflict misses due to accesses to the same cache set: Conflict dominates the picture because two references within the distance range of the cache size access to the same cache set, as shown in the loop 4 and 5 of Figure 2. Possible solutions are array padding or merging.

Patterns in reuse distances: In the reuse distance trace view, each reference is colored with a log value according to reuse distance defined in section 2.2. Cognitively, human discern number of different things well within a dozen, the log_2 value can simplify the reuse distance view with fewer colors. Most importantly, the log_2 value makes the comparison between the reuse distance and the cache size more straightforward.

double a[16384];
double b[128][128];
for (i=0; i<16384; i++)
    a[i] = 1.0;
	

for (i=0; i<16384; i++)
    for (j=0; j<4; j++)
        a[i] = 1.0;
	

for (i=0; i<4; i++)
    for (j=0; j<16384; j++)
      a[j] = 1.0;
	

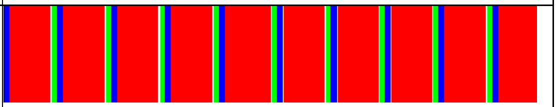
for (i=0; i<16384; i++)
    a[i] = a[i+128];
 	
for (i=0; i<128; i++)
      for (j=0; j<128; j++)
        b[i][j] = b[i+1][j];
	

Figure 2. Given a 1KB directly-mapped cache with 32 bytes line size, the cache miss patterns of different loops are shown in the trace view.

One can select between two types of reuse distance: set and fully associative. Using the set reuse distance one can see the pattern of conflict misses, while the fully associative reuse distances one can see the capacity misses.

For example, the cache miss view shown in figure 3.1 displays 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. The reuse distance view for the matrix multiply is shown in figure 3.3, revealing a strong correlation between the reuse distances and the cache misses.

In figure 3.3, the fully associative cache (FAC) reuse distances of the original 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 locating 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 the picture above.

Correlation traces: Though different feature traces are distributed differently in time, there is often strong correlation between them. For example, the cache miss pattern for different loops can be totally different. Since a programmer wants to apply different optimizing technique to different patterns, it is useful to have the correlation visually revealed.

Each trace view has an interaction that allows the programmer to click on the pixels such that all pixels of the same color as a group are selected. This selection can be passed to another trace view revealing a different feature.

For example, in the loop iteration trace view, a loop nest is selected simply by clicking at a pixel using middle button of the mouse, and in the cache miss view, the corresponding pixels are filtered out, e.g. figure 4.2. On the other hand, clicking a green pixel in the cache miss view will pass the selection of capacity misses to the loop trace view, showing the distribution of capacity miss in different loops, e.g. figure 4.3.

Any two traces can be correlated in a histogram to show the statistically quantity distribution of different patterns. This will be detailed as histogram visualizer in section 3.2.2.

The patterns of any trace correlated to the loop iteration, array name or reference trace views can also be highlighted in the source code, revealing the location of the hot spots. This will be explained as hot-spots visualizer in section 3.2.3.

3.2.3 Histograms view

A histogram is used to analyze the regularity of recurring reference patterns. It is generated from the information of any two trace views T_1 and T_2, for example, let T_1 be a reuse distance trace and T_2 be its corresponding cache miss trace.

The view represents horizontally the patterns in T_1 to analyze (e.g. $\log_2$ reuse distances) and vertically the quantity of identical patterns in trace T_2. In a correlation histogram, the quantity of the correlated trace patterns (e.g. cache misses) is distributed over each pattern of the correlating trace. In this way, for example, one knows exactly how many capacity misses occurs at a certain $\log_2$ reuse distance, how many long distance reuse distances are spatial locality, etc.

Different colors are assigned to the correlated T_2 patterns in the consistent way as its trace view. A legend to the right shows the value of the patterns and their corresponding color.

The chart of the distribution can be sorted either according to the pattern of the correlating trace or according to the quantities of either the total or an individual pattern of the correlated trace. The largest 20 bars can be selected to focus on the most important values. Currently we use bar chart to present the view of a histogram. One can use other charts such as line chart or pie chart.

Here are two example use of the histogram view.

For example, the figure 3.5 shows the histogram of reuse distances with only three larger peaks, indicating that the distance between consecutive memory lines is very regular. In figure 3.5, the histograms of the reuse distances are shown for the original matrix multiplication program. The peaks of the reuse distances indicate that the memory references are periodical. This is also visible in the reuse distance view shown in figure 3.3.

In figure 3.6, the histogram of the reuse distances are shown for the tiled program. 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.

The simple histogram view has been extended to relate the histograms of two different trace patterns. In a correlation histogram, one of the patterns is still shown as the horizontal distribution axis, while the other pattern is shown as color distributions on each bar.

For example, in the locality + reuse distance histogram view of figure 4.5, the reuse distances are distributed along the horizontal axis, while the number of temporal or spatial locality are shown as two colors on each reuse distance bar. The correlation histogram shows that for matrix multiplication, most long reuse distance references are temporal locality. Thus the technique like loop tiling is suitable to exploit the temporal locality.

3.2.3 Hot spots view

The program hot spot view lists the source program in a text editor that allows highlighting. The visualized information can indicate the location of the bottlenecks.

First, the locations of the array reference and loop iteration are annotated to the address trace by the profiler. Then the correlation histogram that correlates any trace pattern to the reference or loop trace pattern can generate a series of hot spot views, indicating the relative importance of individual array reference or loop nest.

For example, the cache miss trace correlated to array reference trace yields four hot spot views, for hits, compulsory, conflict and capacity respectively. In figure 4.4, for example, the most conflicted missed references are shown as hot spots. They are mostly located in a single loop nest.

The reuse distance trace correlated to loop iteration trace yields a number of hot spot views, indicating the loop with better locality (short reuse distance hot spots) or worse (long reuse distance hot spots).

4 Results

The approach has been applied to many examples [ppt]. To look is to believe. In this section, the approach explained in Section 3 will be applied in several experiments to see how effective cache performance optimizations are selected by using this tool.

4.1 Tiling of perfectly nested loop

As a simple example, first consider a matrix multiplication program.

```
      DO 10 i=1,N
       DO 10 j=1,N
        c(i,j)=0
        DO 10 k=1,N
10        c(i,j)=c(i,j)+a(i,k)*b(k,j)
```

The instrumentation was done using the FPT compiler [edh98].To demonstrate, the program is scaled down to N=40. Using a small size of the program work set, a smaller cache is also configured to a 1KB direct-mapped cache with 32 byte line size. In the experiment, a small work set and cache will let the cache simulation be faster, but still revealing the fact of the cache behavior of the program.

The cache miss patterns are shown in figure 3.1.

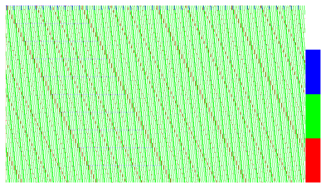
The cache suffers 33.9% misses. Cold misses are shown in blue, mostly at beginning of the execution (in the upper left). As can be expected, the beginning of the program suffers cold misses more frequently to arrays A and B. Due to the intermittent access to matrix C, however, a few cold misses happen occasionally till the end of the trace (slanted downward line in the figure).

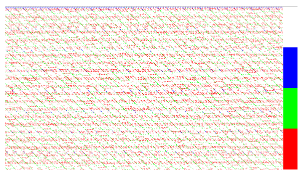
The capacity misses are shown densely in green. This pattern comprises most of the cache misses. The large number of capacity misses may indicate that the cache size is relatively small. The pattern looks rather regular, since matrix B is thrown out of the cache by each “I-loop” iteration.

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

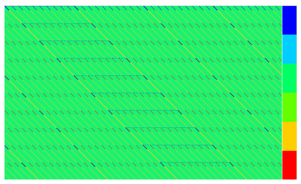
In the $\log_2$ reuse distance trace view shown as figure 3.3, one can see that many reuse distance are larger than the cache size.

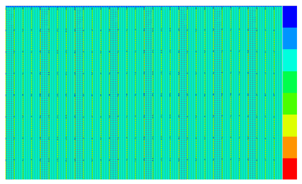
In the corresponding reuse distance histogram view shown as Figure 3.5, the quantity of reuse distance larger than the cache size is clear. This phenomenon suggests loop tiling or blocking as a possible solution.


(1) original program.

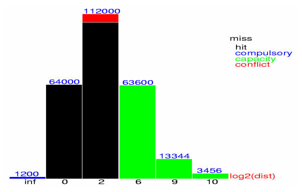

(2) tiled program

The cache misses trace view Blue=cold, Green=capacity, Red=conflict misses

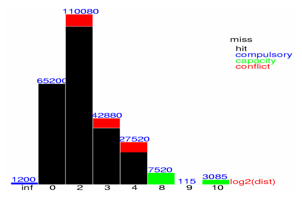

(3) original program


(4) tiled program

The $\log_2$ FAC reuse distance view:
similar pattern correlated to the cache miss trace view.



(5) original program.



(6) tiled program

The histogram view of the $\log_2$ reuse distances. `inf' corresponds to the compulsory miss.

Figure 3. Visualizing the matrix multiplication.

The matrix multiplication loop iteration space is tiled into size S x S blocks with respect to the column-major order in Fortran [wolf91]. The tiled loop is shown as follows.

```
     DO 10 i=1,N
      DO 10 j=1,N
10    c(j,i)=0
     DO 20 k'=1,N,S
      DO 20 i'=1,N,S
       DO 20 j=1,N
        DO 20 k=k', min(k'+S-1,N)
         DO 20 i=i', min(i'+S-1,N)
20        c(j,i)=c(j,i)+a(j,k)*b(k,i)
```

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.

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 the cache miss is visible in figure 3.2. Here a tile of (5x5) is selected.

The comparison with the original view on the left shows that loop tiling does reduce the number of long reuse distances references. As expected, the number of compulsory miss doesn't change because the whole work set remains the same. But one can see that compulsory miss spread more over the whole time because new tile of data is brought into the cache not only from the beginning. Due to the new shape of the array, more conflict misses occurs in the tiled trace. In total, the number of cache misses is reduced.

The improvement over the original version is shown in table 1.

Table 1. Cache misses reduced by tiling

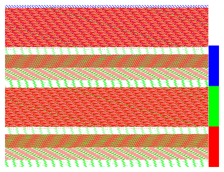
number of	Original	Ratio	tiled	Ratio
Refers	257600	100.0%	257600	100.0%
Compulsory	1200	0.5%	1200	0.5%
Capacity	80400	31.2%	10720	4.2%
Conflict	5660	2.2%	20726	8.0%
Misses	87260	33.9%	32646	12.7%

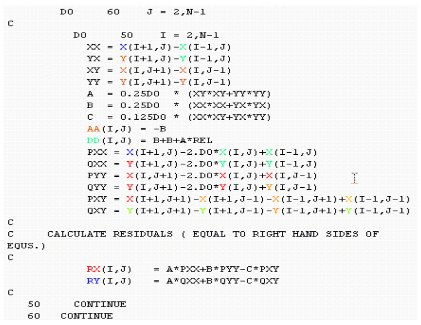
4.1 Array padding

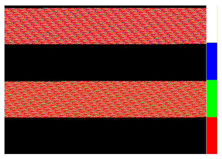
The next example to demonstrate is TOMCATV, one of the SPECfp95 benchmark programs. The array declarations and the main loop nests are shown as the following.

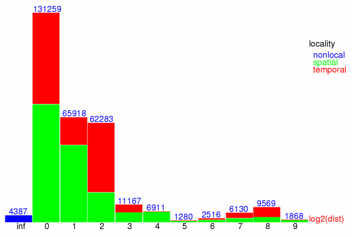
```
PARAMETER (nmax=513)
...
REAL*8 aa(nmax,nmax),dd(nmax,nmax),d(nmax,nmax)
REAL*8 x(nmax,nmax),y(nmax,nmax)
REAL*8 rx(nmax,nmax),ray(nmax,nmax)
...
DO iter = 1,intact
 ...
 DO j = 2,n-1
   DO i = 2,n-1
 xx=x(i+1,j)-x(i-1,j)
 yx=y(i+1,j)-y(i-1,j)
 xy=x(i,j+1)-x(i,j-1)
 yy=y(i,j+1)-y(i,j-1)
     ...
```

The instrumentation and cache simulation are done in the same way as in the matrix multiplication.

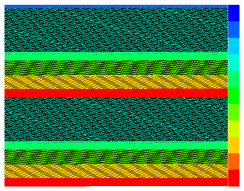

(1)Cache miss trace of the whole program.


```
        DO      60     J = 2,N-1
C
          DO      50     I = 2,N-1
            XX = X(I+1,J)-X(I-1,J)
            YX = Y(I+1,J)-Y(I-1,J)
            XY = X(I,J+1)-X(I,J-1)
            YY = Y(I,J+1)-Y(I,J-1)
            A  = 0.25D0  *  (XY*XY+YY*YY)
            B  = 0.25D0  *  (XX*XX+YX*YX)
            C  = 0.125D0 *  (XX*XY+YX*YY)
            AA(I,J)  = -B
            DD(I,J)  = B+B+A*REL
            PXX  = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
            QXX  = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
            PYY  = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
            QYY  = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
            PXY  = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
            QXY  = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
C
C      CALCULATE RESIDUALS ( EQUAL TO RIGHT HAND SIDES OF
EQUS.)
C
            RX(I,J)     = A*PXX+B*PYY-C*PXY
            RY(I,J)     = A*QXX+B*QYY-C*QXY
C
   50     CONTINUE
   60   CONTINUE
```
(4) Hot spots of conflict misses in the source.


(2) Cache miss trace of the second loop nest.



(5) Classify locality in reuse distances.


(3) Iteration trace filtered the conflicts miss as black

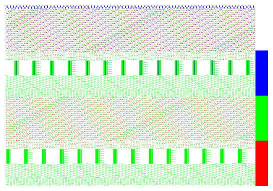

(6) Cache miss trace of the padded program.

Figure 4. Visualizing the TOMCATV benchmark program

The input size of the program is changed to *itact=2* and *n=50*. It is still simulated as on a *1* KB direct-mapped cache.

The visualized cache misses are shown in figure 4.1. There are a lot of conflict misses in this view. In the following, we can see how the hot spots in the source code are traced. The conflict misses are intensified in particular bands in figure 4.2, which are corresponding to the second loop nest in figure 4.3. This leads to focus the attention to the most important loop nest. Here one sees the use of the correlation traces between cache misses and loop iterations.

In figure 4.4, the hot spots of conflict misses are visualized by highlighting the corresponding array references in the source code. Here one sees the use of the correlation between cache misses and the array references.

To choose a suitable technique to improve the program, the programmer can refer to the classification of the reuse distances as temporal or spatial localities. In figure 4.5, there are more spatial localities. This indicates data transformation may be useful to remove spatial localities.

In the program, parameter nmax is set to *513*, which is very close to the power of 2. For example, the two dimensions for arrays *X* are nmax, thus the array references *x(i+1, j)* and *x(i, j+1)* cause conflict miss in directed-mapped cache because they address to the same cache line. The same problem happens to arrays *Y, RX, RAY*, etc.

To avoid the conflicted cache lines, the arrays are padded with empty addresses by changing nmax to *524*. The cache misses view of the padded TOMCATV is shown in figure 4.6. After array padding, most conflict misses are gone.

The improvement over the original version is shown in table 2. The cold and capacity misses increases slightly due to the increase of work set by changing dimension from *513* to *524*. Conflict misses are reduced from *43.8%* to *4.1%* such that the overall cache miss ratio is reduced from *56.7%* to *17.2%*.

Table 2. Cache misses reduced by array padding for TOMCATV benchmark where itact=2, n=50.

number of	Original	Ratio	Tiled	Ratio
Refers	303288	100.0%	303288	100.0%
Compulsory	4387	1.4%	4423	1.5%
Capacity	34766	11.5%	35291	11.6%
Conflict	132876	43.8%	12388	4.1%
Misses	172029	56.7%	52102	17.2%

5 Related work

There are many tools to measure the cache miss ratio, either through cache simulation [uhlig97], such as Dinero [hill89], Cprof [lebeck94] or sampling from hardware counters like VTune [atkins96]. There are also compiler techniques to estimate the cache miss ratio analytically [ghosh99, harper99].

In order minimize cache misses, loop and data transformations to improve the data locality [mckinley99, lebeck94, wolf91], such as loop tiling/blocking, loop fusion, array merging, packing, padding and data 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 [vanderdeijl97] 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 [bosch00] visualizes cache behavior through statistical histograms of the cache lines. The histograms show that 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 is relatively small in comparison to the number of memory references of a program. However, this doesn't allow 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.

6 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

[atkins96] M. ATKINS and R. SUBRAMANIAM. PC software performance tuning. Computer, 29(8):47, Aug 1996.

[belady66] L. A. BELADY. A study of replacement algorithms for a virtual storage computer. IBM Systems Journal, 5(2):78--101, 1966.

[bosch00] 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.

[ppt] E. D'HOLLANDER, Y. YU, and K. BEYLS. Parallel Programming Tools. http://elis.rug.ac.be/paris/ppt, 2001.

[edh98] E. D'HOLLANDER, F. ZHANG, and Q. WANG. The fortran parallel transformer and its programming environment. Journal of Information Sciences, 106:293--317, 1998.

[ghosh99] S. GHOSH, M. MARTONOSI, and S. MALIK. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems, 21(4):703--746, Jul 1999.

[harper99] J.Harper, D.Kerbyson, and G.Nudd. Analytical modeling of set-associative cache behavior. IEEE Transactions on Computers, 48(10):1009--1024, Oct 1999.

[hill89] M. HILL and A. SMITH. Evaluating associativity in {CPU} caches. IEEE Transactions on Computers}, 38(12):1612--1630, Dec 1989.

[lebeck94] A.R. LEBECK and D.A. WOOD. Cache profiling and the {SPEC} benchmarks: {A} case study. Computer, 27(10):15--26, Oct. 1994.

[mckinley99] K. MCKINLEY and O. TEMAM. Quantifying loop nest locality using (SPEC95) and the Perfect benchmarks. ACM Transactions on Computer Systems, 17(4):288--336, 1999.

[uhlig97] R. UHLIG and T. MUDGE. Trace-driven memory simulation: A survey. ACM Computing Surveys, 29(2):128--170, Jun 1997.

[vanderdeijl97] E. VANDERDEIJL, G. KANBIER, O. TEMAM, and E. GRANSTON. A Cache Visualization Tool. Computer, 30(7):71--78, Jul 1997.

[wolf91] M.E. WOLF and M.S. LAM. A data locality optimizing algorithm. In: Proceedings of the {ACM} {SIGPLAN} '91 Conference on Programming Language Design and Implementation ({PLDI})}, volume 26, pages 30--44, 1991.

Yijun Yu obtained BS and MS degrees from the CS Department of Fudan university, China in 1992 and 1995 respectively, majored in software engineering. In 1998, he obtained a PhD from the Institute of Parallel Processing in Fudan University. In 1999, he joined the PARIS research group in Ghent University as a postdoctoral researcher. Since 2003, he is a research associate in the CS department of the University of Toronto. His research interests include performance visualization, optimizing compilers, parallel programming tools, software re-engineering, data migrations and web-services.

Kristof Beyls is a PhD student at University of Ghent, Belgium. He obtained a MS degree from the CS department at the same university in 1999. His research interests include optimizing compilers, cache performance, parallel processing and performance debugging.

Erik H. D'Hollander} graduated from the universities of Ghent (EE) and Louvain (CS). He did research on multiprocessors for continuous system simulation at the Computer Science Department of the UCLA (1979-1980) and obtained a PhD from the University of Ghent in 1980. As a member of the Parallel Information Systems group his research interests focus on optimizing compilation techniques for parallel and embedded computer architectures.

RUG - Dept. of Electrical Engineering Parallel Information Systems, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium, dhollander@elis.ugent.be.