A Cache Visualization Tool

Improved cache performance is crucial to improved code performance. By visualizing cache behavior as the program is simulated, this memory analysis tool dynamically represents cache phenomena. The results can guide developers in making better software and hardware optimizations.

Eric van der Deijl and Gerco Kanbier
Leiden University

Olivier Temam
Versailles University

Elena D. Granston
Rice University

Cache performance strongly influences the overall performance of software. As a result, cache analysis and optimization continue to be actively researched. Typically, cache simulators provide only raw, global information such as overall cache miss ratio. To improve cache performance, we need to better understand cache behavior through, for example, the impact of software optimizations on cache performance and the behavior of new hardware cache designs.

It can be hard to anticipate a code cache behavior on the basis of its source code. This is because data are mapped into cache with a mapping function that performs an arithmetic operation, a modulo on the data address. Also, the base addresses of data structures like arrays are not specified by the programmer; therefore, the relative cache positions and interactions of these data structures are normally not controlled. Finally, it can be difficult to apprehend the interactions in cache of many different memory references.

Cache behavior analysis can be viewed as a two-step process. First, you must identify code sections with poor cache performance. Second, you must understand the causes for poor performance in these code sections. Cache profilers are used for the first task. For example, MemSpy[1] provides memory performance statistics at the procedure level, Mtool[2] at the procedure or basic block level, and PFC-Sim[3] at the statement level. CProf[4] provides cache statistics at the statement level and also classifies the different misses. The Cache Visualization Tool's (CVT's) purpose is to address the second task, understanding the causes of poor cache performance. It is thus a complement to cache profilers.

Cache behavior can be accurately analyzed by *visualizing* what happens in cache for given types of code. For this purpose, we developed a CVT, which both dynamically visualizes cache content and provides related statistics.

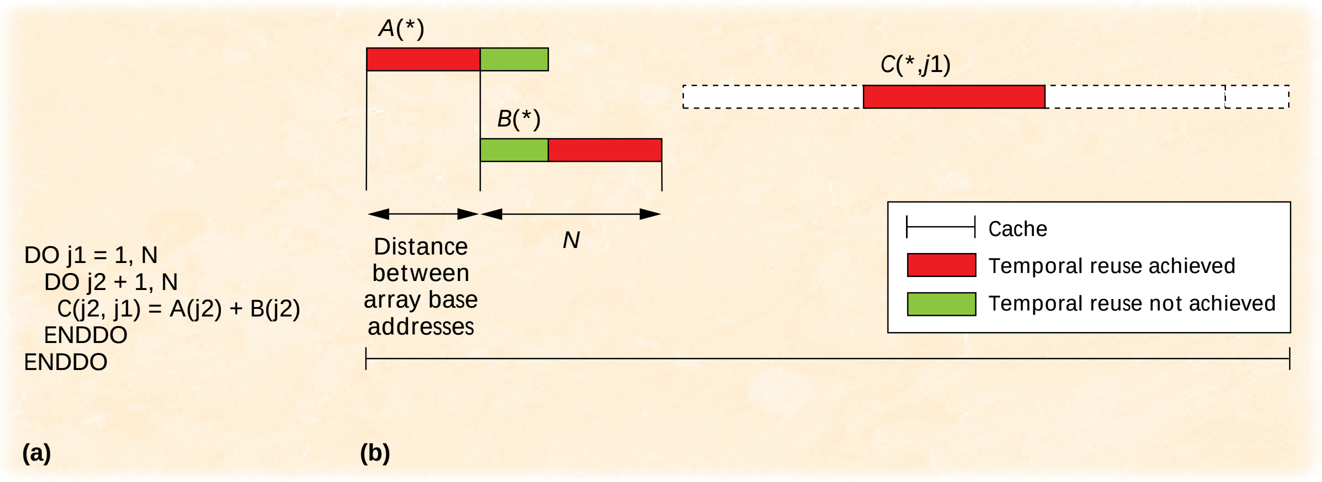
OVERVIEW

The CVT is a graphical X Windows tool based on Motif. The main window is a grid representing the cache content. A cache is composed of cache blocks or cache lines (a set of words with consecutive addresses), and each box in the grid corresponds to a cache line. The lines are ordered from top to bottom and left to right. Line 0 is at the top left, and the last line is at the bottom right. During execution, the CVT colors the cache lines according to the array or the array reference that last used it. A command panel in the main window lets you control the execution. Another window displays various statistics (updated dynamically) on the cache, the arrays, the array references, and the program counters. You can feed a CVT with Fortran loop nests (nested instructions that repeat until a condition is met) in a specific format or with different kinds of program traces. Finally, a CVT has a built-in cache simulator, but you can plug in other cache simulators to, for instance, study novel cache designs.

The underlying principle of cache is to exploit the intrinsic temporal locality (reuse of data within short time intervals) and spatial locality (accesses to data with close addresses) present in most codes. A miss occurs when data required by the processor are not present in cache. Though cache performance are best measured by the average time to access data, the number of cache misses is a good indication of how efficiently the cache is used. Three types of cache misses are usually distinguished: *compulsory* (first access to data), *capacity* (cache size is too small to store enough data) and *conflict*, or interference, miss (conflicts due to the cache mapping function). The CVT is particularly useful in understanding how cache conflicts (which can be complex) work.

To understand how cache conflicts occur, visualizing cache operations is most helpful. Figure 1 on the next page shows the layout of arrays A, B, and C in cache for a complete iteration of loop $j1$. The cache is represented at the bottom of the figure as a horizontal line. Mapping an array element to cache can be viewed as projecting the array element vertically to the cache (the cache is assumed to be direct-mapped). So two data with the same horizontal position map into the same cache location, thereby creating a conflict.

Assuming the cache is large enough to contain the $2N$ words of arrays A and B, no capacity miss would occur. On the other hand, depending on the base addresses of A and B, these two arrays may overlap in cache, inducing conflict misses; thus, part of the reuse of A and B cannot be achieved. In Figure 1, when the rel-



```
DO j1 = 1, N
   DO j2 + 1, N
     C(j2, j1) = A(j2) + B(j2)
   ENDDO
ENDDO
```

A(*)
B(*)
C(*,j1)
Distance between array base addresses
N
Cache
🟥 Temporal reuse achieved
🟩 Temporal reuse not achieved
(a)
(b)

Figure 1. Example of (a) a Fortran loop nest and (b) its cache layout.

ative cache distance between the array base address of *A* and *B* is smaller than *N*, both arrays overlap and flush each other from cache on each iteration of *j*1.

CVT can be used for the following tasks:

- *Detailed analysis of software cache optimizations.* CVT can visualize the effects of loop transformations (including techniques such as tiling[5] and blocking[6,7]) and uncover potential hazards.
- *Evaluation and debugging of hardware cache optimizations.* To validate a new cache design, the usual strategy is to simulate several benchmarks and compare global miss ratios. If the design does not perform as expected, it is difficult to understand the results with only a cache simulator. What you need is a *cache debugger* to monitor the changes in cache at each step.
- *Detailed cache performance analysis of code constructs.* Profiling tools identify the routines, loop nests, or statements that cause most cache misses. CVT is much slower than profilers because of visualization. It should therefore be used only in the final analysis step once you have identified, with a cache profiler, code constructs that cause the most misses.

CVT COMPONENTS

CVT's three main components are execution control, input, and statistics.

Execution control

You can feed CVT with a memory address trace, either collected from a program or generated in CVT. For each new memory reference in the trace, CVT colors the corresponding cache line according to either the load/store instruction that accessed it or the data structure contained in it.

In some respects, you might consider CVT a cache debugger, which uses an address trace in place of an instruction trace. Execution controls share similarities with a program debugger, like dbx. Naturally, the trace can be run, though visualization requires that its speed be adjustable. Visualization tends to slow trace execution considerably, so we implemented *skip forward* and *skip backward* functions to quickly browse through the trace. Also, as in debuggers, CVT uses breakpoints, and we implemented several kinds: as a function of the number of memory references, array references used, array sections referenced, cache area used, and loop indices.

Input

CVT accepts three different types of input—loop nests, object code traces, and source code traces—each of which trades off accuracy, ease of use, and scope (the nature of code that can be visualized).

Loop nests. We provide only the essential parameters to generate the array address trace in CVT: loop boundaries, array base addresses, and the array references' subscripts' coefficients to generate the array address trace. This format sacrifices accuracy and scope (only linear array subscripts can be used) in exchange for ease of use. With this format, we can easily backtrack in the loop, modify array base addresses (which play a significant role in several interference phenomena), and other loop parameters.

Object code traces. To improve accuracy, you can also feed traditional object code traces to the CVT, using tracers like Spy. For each load/store instruction, CVT needs only the instruction address, data address, and read/write instruction. Because CVT has no information on the code's data structures, it uses load/store instruction addresses to color the trace. Such traces extend CVT's scope to non-numerical codes.

Source code traces. These traces fall between CVT loops and object code traces in terms of accuracy, scope, and ease of use. They are obtained by directly instrumenting memory references in the source code, like array references in Fortran. Such traces are less accurate than object code traces because compiler optimizations are ignored. However, source traces mean that CVT can be applied to many codes and complex memory references, such as indirectly addressed arrays in sparse codes. We are currently working on modifying Gnu Fortran-77 to provide similar information for each load/store reference without any modification or instrumentation of the source code.

Statistics

CVT supplies, and dynamically updates, statistics

```
DO i = 0,99
 DO j = 0,99
    X(j, i) = X(j, i + 1) + Y(j, i)
 ENDDO
ENDDO
```

(a)

during visualization. These statistics either concern data or instructions (for example, arrays, load/store references, and whole trace) or cache behavior (per cache line and per cache area, among others). Statistics include misses, references, and reuses, and can be displayed in different forms such as ratios, cumulated, and noncumulated.

SAMPLE APPLICATIONS

Our examples involve visualizing cache interference, cache behavior of blocking, performance evaluation of complex loops, hardware optimization, and locality analysis. The cache size was 8,192 bytes, the line size was 32 bytes, and the cache was direct-mapped. These parameters correspond to Digital Equipment Corporation's Alpha 21164 on-chip first-level data cache. All Fortran code featured single-precision.

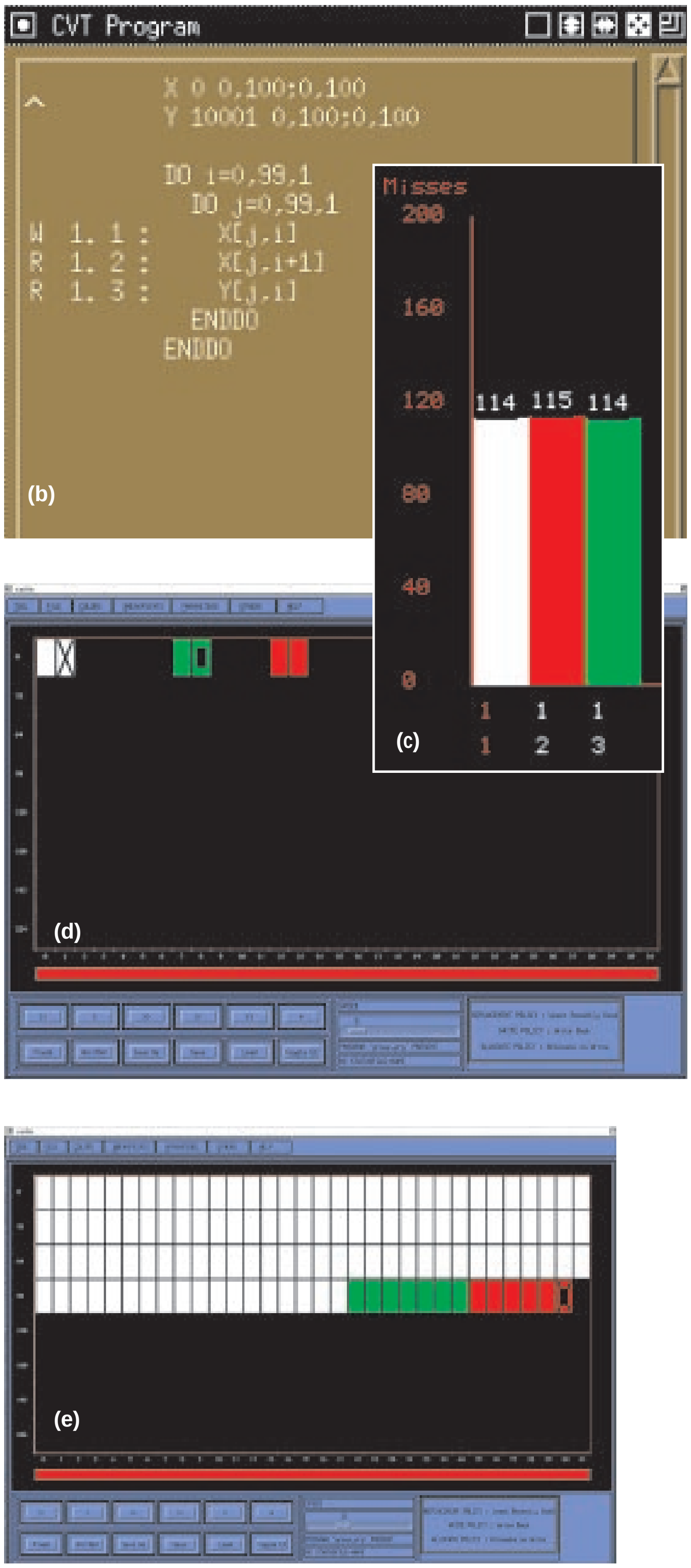
Visualizing cache interference

Conflict misses are a major source of cache misses.[8] However, unlike capacity misses, cache interference phenomena can be difficult to detect at the source code level because their occurrence pattern is usually more complex.[9] Also, the parameters that strongly influence the occurrence of cache conflicts, such as array base addresses, are not available at the source code level.

Figure 2 shows how CVT can visualize the occurrence of cache interference phenomena and help remove cache conflicts. Figure 2a is the original Fortran code; Figure 2b shows the code as it is provided to CVT. Figure 2c shows the number of misses for the three array references to the two arrays (*X* and *Y*) in the loop nest, mapping colors, array references, and reference numbers. References are numbered as follows: *Y*(*j*, *i*) is reference 1.3 and is green (that is, its statement id = 1 and its array reference id = 3), *X*(*j*, *i*) is reference 1.1 and is white, and *X*(*j*, *i* + 1) is reference 1.2 and is red. In the figure, a dependence between *X*(*j*, *i*) and *X*(*j*, *i* + 1) means that *X*(*j*, *i*) normally reuses most of the data referenced by *X*(*j*, *i* + 1). (This reuse has been called *group-temporal reuse*.[5]) However, if the relative base addresses of arrays *X* and *Y* are such that *Y*(*j*, *i*) is located between *X*(*j*, *i*) and *X*(*j*, *i* + 1), no reuse can occur because *Y*(*j*, *i*) flushes elements of *X*(*j*, *i* + 1) before they can be reused by *X*(*j*, *i*). This phenomenon is difficult to determine from the source code, but CVT can visualize it.

For example, Figure 2d shows *Y*(*j*, *i*), which is red, located between the two other references, which are white and green. The small black square in the green box corresponds to the cursor; it is positioned within the line where the last reference occurred.

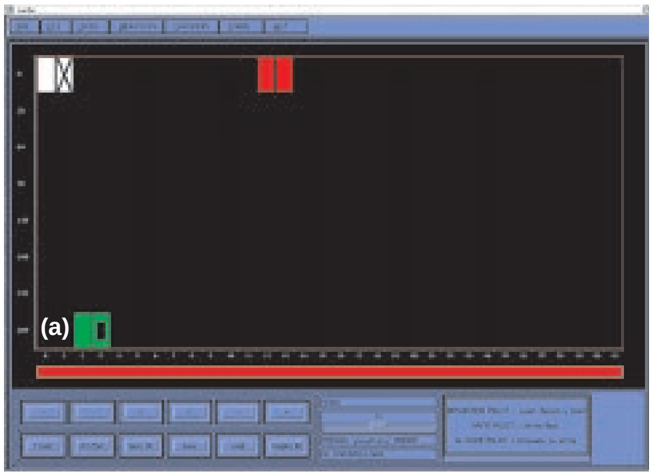
Figure 2d shows that the program just began exe-

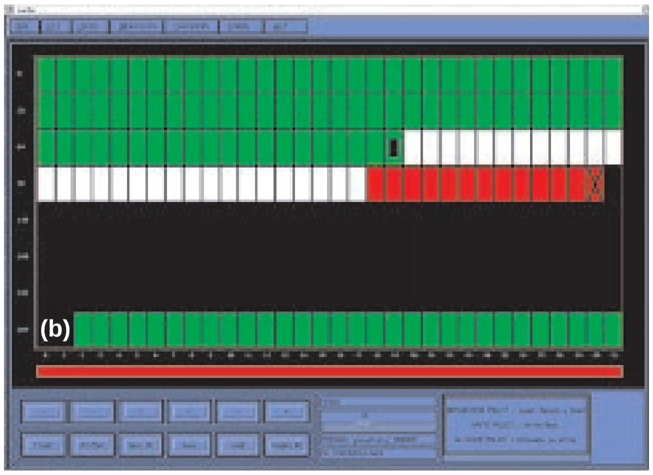


```
X 0 0,100:0,100
Y 10001 0,100:0,100

DO i=0,99,1
  DO j=0,99,1
W 1. 1 :   X[j,i]
R 1. 2 :   X[j,i+1]
R 1. 3 :   Y[j,i]
  ENDDO
ENDDO
```

Figure 2. Cross-interference: (a) Fortran code; (b) CVT code; (c) number of misses per array reference; (d) visualization after eight iterations of **j***; (e) visualization after eight iterations of* **i**.





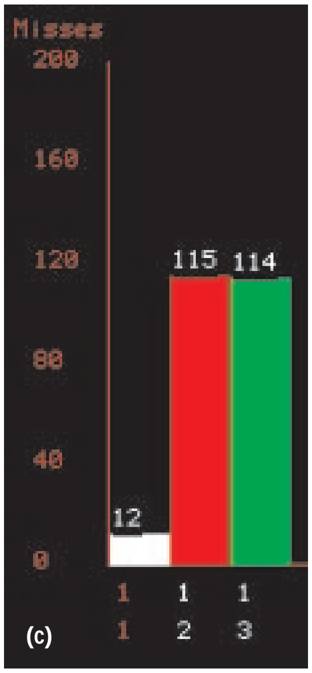


Figure 3. No cross-interference: (a) visualization after eight iterations of **j***; (b) visualization after eight iterations of* **i***; (c) number of misses per array reference.*

cuting, with only eight iterations of loop *j*. In Figure 2e, the program executed eight iterations of loop *i*.

References are moving left to right so that reference *Y*(*j*, *i*) progressively overlaps with reference *X*(*j*, *i* + 1)—reference 1.2—preventing any reuse by reference *X*(*j*, *i*), or reference 1.1. The statistics window in Figure 2c confirms that all references benefit only from spatial reuse—almost the same number of misses occur for *X*(*j*, *i*) and *X*(*j*, *i* + 1). Figure 3 shows CVT visualization, too, only this time we moved reference *Y*(*j*, *i*) out of the interval created by the two other references. Figure 3a resembles Figure 2d, except that *Y*(*j*, *i*) is not located between *X*(*j*, *i*) and *X*(*j*, *i* + 1). Figure 3b shows this after eight iterations of loop *i*. Now reference *Y*(*j*, *i*) does not interfere with the reuse of *X*(*j*, *i* + 1) by *X*(*j*, *i*). The statistics window in Figure 3c confirms that the number of misses incurred by reference 1.1, corresponding to *X*(*j*, *i*), is low, indicating temporal reuse, while the two other references exhibit only spatial reuse. We have thus achieved an optimization that removes all interference misses.

Cache behavior of blocking

Research has shown[6] that with changes in matrix dimension or block size values, the cache miss ratio of matrix multiplication fluctuates because of self-interferences (distinct parts of the same array flush each other from cache). In the blocked version of matrix multiplication below, we will vary the matrix dimension with *B* = 30.

```
DO kk = 0, N-1, B
  DO jj = 0, N-1, B
    DO i = 0, N-1
      DO k = kk, min(kk + B-1, N-1)
        reg = X(k, i)
          DO j = jj, min(jj + B-1,
            N-1)
              Z(j, i) = Z(j, i)
                + reg * Y(j,k)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
```

In this loop, array *Y* can fall victim to self-interference. When *N* = 224, the cache layout of this array after four iterations of loop *i* is shown in Figure 4a. Note that the block of *B* × *B* elements of *Y* is laid out in cache after a single iteration of loop *i*, and it is normally reused in subsequent iterations.

Coloring by array is used for the CVT windows of Figures 4a and 5a. White is for array *X*, red for array *Y*, and green for array *Z*. In the statistics windows of these figures, misses are shown for all references as follows: white for the read reference to *X*(*k*, *i*); red, read reference to *Y*(*j*, *k*); green, read reference to *Z*(*j*, *i*); blue, write reference to *Z*(*j*, *i*).

In Figure 5, where *N* = 256, Figure 5a shows the cache layout of *Y* after four iterations of loop *i*. The same memory references have been performed in both cases, since both loops *k* and *j* perform *B* = 30 iterations for the first four iterations of loop *i*. The comparison between Figures 4 and 5 shows that for *N* = 256, array *Y* occupies less of the cache, suggesting that some elements of the *Y* block overlap with each other.

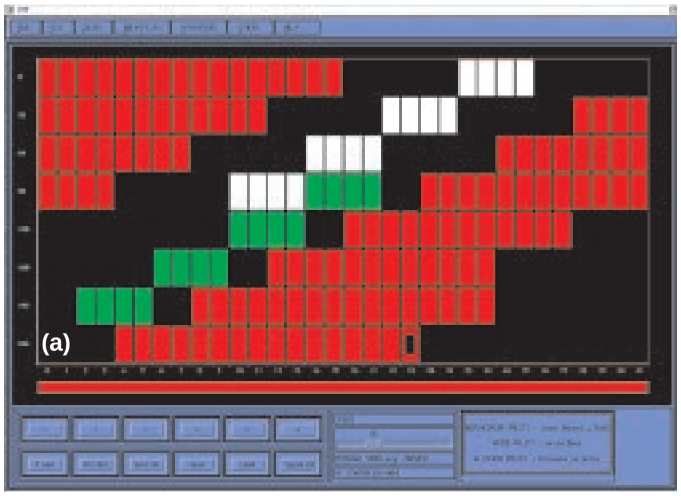
Figures 4b and 5b, respectively, confirm this by showing the number of misses per references for *N* = 224 and *N* = 256. For *N* = 224, array *Y* has 120 misses; for *N* = 256, array *Y* has 480 misses. The cache achieves only spatial reuse of the *Y* block, and it must reload the block on each iteration of *i*.

When running the CVT for *N* = 256, the absence of reuse for *Y* clearly appears because each time the cursor moves to a new line of block *Y*, CVT draws a cross indicating a cache miss. In contrast, when *N* = 224 this does not occur.

Performance evaluation of complex loops

As the number of references in a loop increases, potential interactions between array references proliferate, complicating loop analysis.

We extracted the loop nest below from the `EFLUX` routine of Perfect Club code `FLO52`.[10] Array subscripts



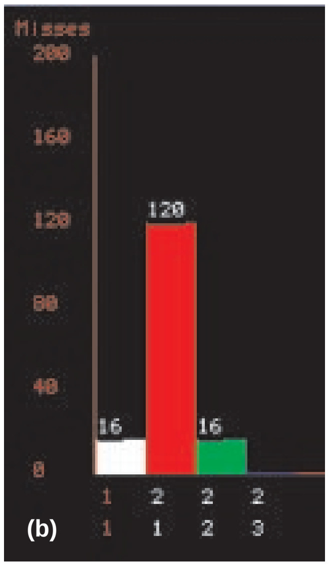
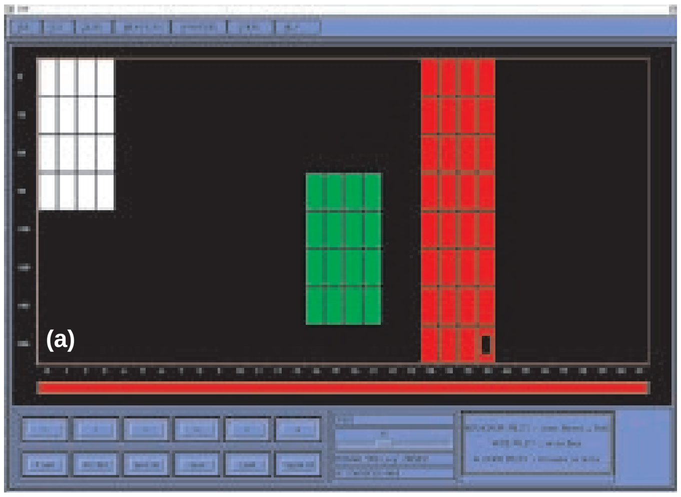


Figure 4. No self-interference: (a) visualization after four iterations of i*; (b) number of misses per array reference.*



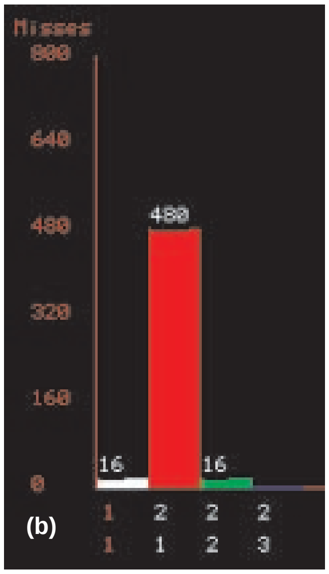


Figure 5. Self-interference: (a) after four iterations of i*; (b) number of misses per array reference.*

appear to differ by only a few constants, suggesting that the relative cache distance is constant between any two array references. However, because array declarations are distinct, the linear expression of references may differ so that the relative distance between any two array references is not necessarily constant, and array references may conflict irregularly.

```
DO 10 J = 2, JL
 DO 10 I = 1, IL
    XY   = X(I, J, 1) -X(I, J - 1, 1)
    YY   = X(I, J, 2) -X(I, J - 1, 2)
    PA   = P(I + 1, J) +P(I, J)
    QSP  = (YY * W(I + 1, J, 2) -XY *
       W(I + 1, J, 3))/W(1 + 1, J, 1)
    QXM  = (YY * W(I, J, 2) -XY *
       W(1, J, 3))/W(I, J, 1)
    FS(I, J, 1)= QSP * W(I + 1, J, 1)
       +QSM * W(I, J, 1)
    FS(I, J, 2)= QSP * W(I + 1, J, 2)
       +QSM * W(I, J, 2) +YY * PA
    FS(I, J, 3)= QSP * W(I + 1, J, 3)
       +QSM * W(I, J, 3) -XY * PA
   FS(I, J, 4)= QSP * W(I + 1, J, 4)
      +P(I + 1, J)) + QSM *(W(I, J, 4)
      +P(I, J))
  10 CONTINUE
```

To analyze the cache behavior of this loop nest, we ran a source trace on the CVT. Figure 6 on the next page shows the cache miss distribution.

Misses peak between cache lines 114 and 118. By running the visualization again we found this peak occurs between the 3,700th and the 4,400th reference. When we looked at the array reference misses, we found a strong increase of cache misses in this time interval for references 11.0 (the first dark blue bar in Figure 7a) and 14.0 (the second gold bar in Figure 7b).

When we run the corresponding trace section, Figures 7c and 7d show two consecutive references to

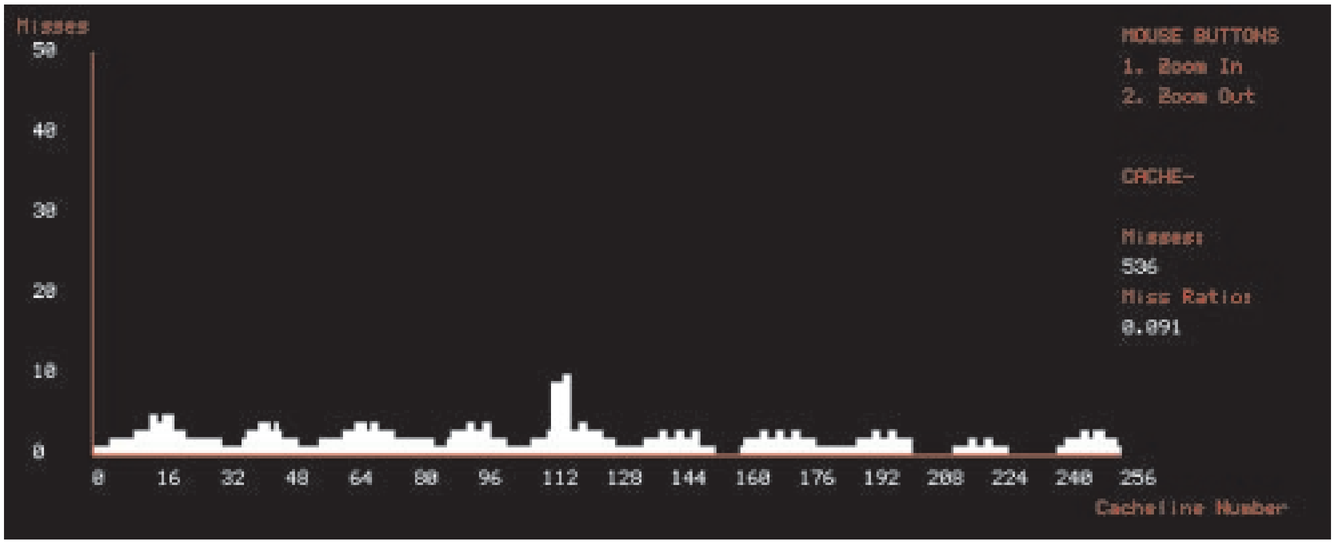
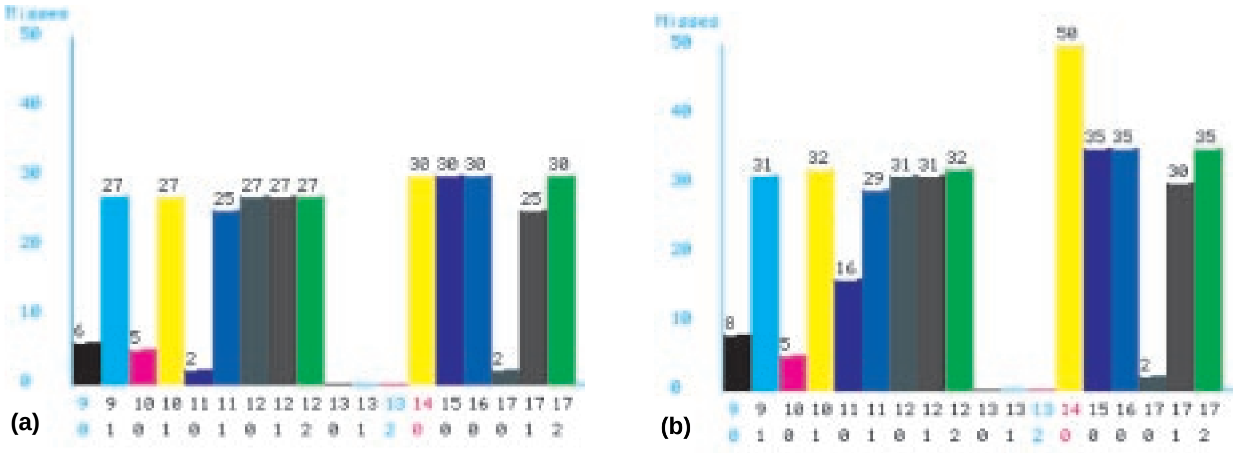


Figure 6. Cache miss distribution.



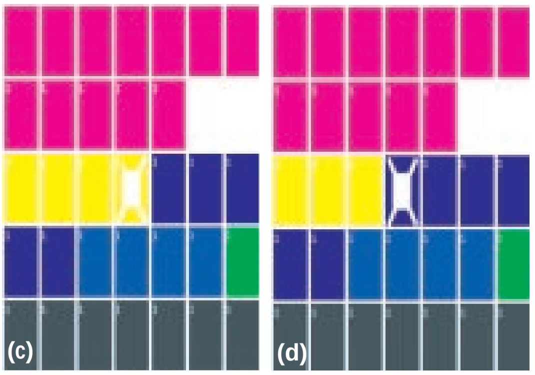


Figure 7. Analysis of loop nests with multiple references: (a) number of misses per array reference before the cache phenomenon; and (b) number of misses per array reference after the cache phenomenon; (c) and (d) cache content for two consecutive uses of cache line 116.

cache line 116 and neighboring lines, colored by array references. A miss occurred in line 116 for reference 14.0 (gold) and 11.0 (dark blue).

Because the same phenomenon occurs for several consecutive references to this cache line and the neighboring ones, we know that references 11.0 and 14.0 are competing for the same cache lines. This is spatial interference, also called a *ping-pong* effect. With the array reference ids, we could find the corresponding array references (respectively *p*(*i*, *j*) and *fs*(*i*, *j*, 1)) in the source code. To remove the cache phenomenon, we copied reference 11.0 (p(*i*, *j*)) to a temporary array with the same declaration as *fs*.

This particular case is difficult to pinpoint and more difficult to predict. The interval between these two references decreases until they fall into the same cache line. Then, the references spread and do not ping-pong anymore.

To our surprise, we found that such cases occur in many loops. For example, strong spatial interference also occurs in blocked matrix multiplication even though arrays do not compete initially for the same cache lines. Such "periodic" ping-pong phenomena can correspond to a significant share of all cache conflicts.

Hardware optimization

Cache visualization can also help us compare the behavior of two cache architectures on selected code segments. For example, Qiang Yang and Lan W. Yang proposed a direct-mapped cache size of 2^n-1 to avoid mapping conflicts resulting from power-of-two array dimensions.[11] The 2^n-1 cache size modifies the mapping function, thus creating a hashing effect. We investigated several variations on this idea, specifically one with a cache size of 2^n-3. We studied the effect of this cache design on algorithms known to be sensitive to cache

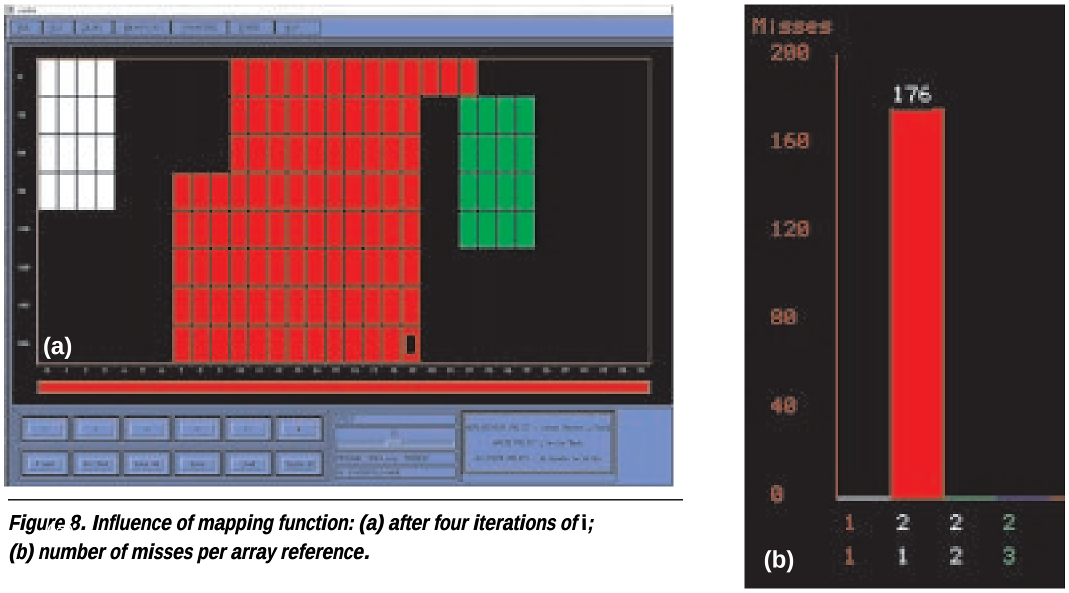


Figure 8. Influence of mapping function: (a) after four iterations of `i`*; (b) number of misses per array reference.*

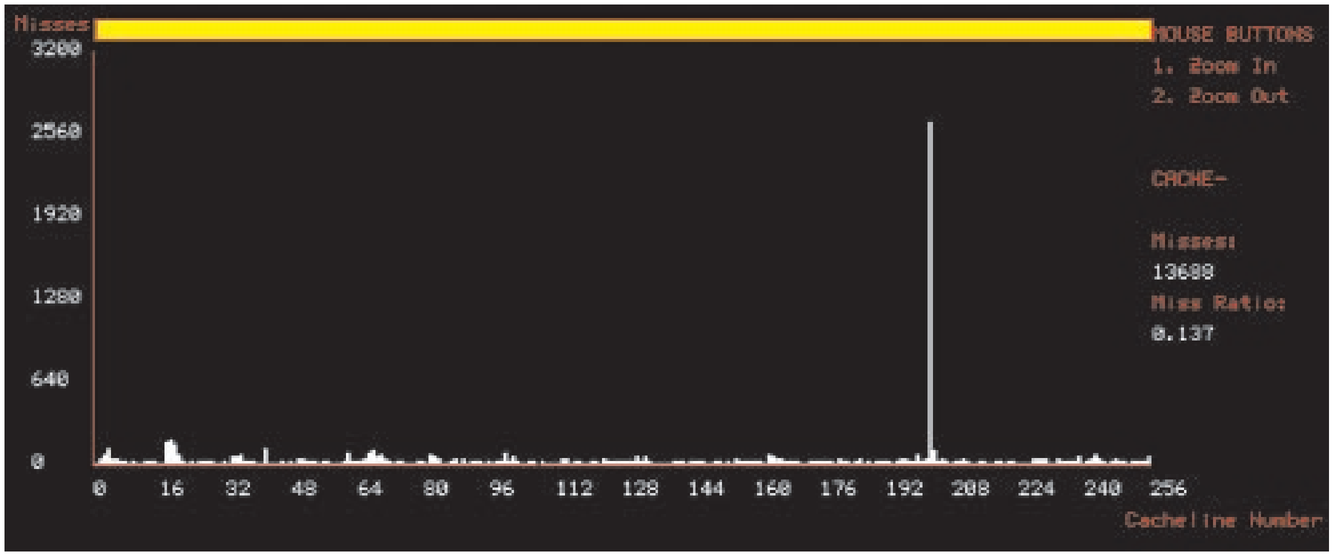


Figure 9. Cache miss distribution for compress.

interference, notably blocked matrix multiplication.

In our earlier discussion on blocking, the elements of matrix *Y* piled up in a small section of cache, inducing self-interference and preventing temporal reuse. Figure 8a shows the CVT visualization of the same loop used in Figures 4 and 5, with four iterations for the 2^n-3 cache, which was a direct-mapped cache with 253 lines of 32 bytes each. Figure 8 clearly shows that array *Y* used more of the cache, which reduced self-interference and increased temporal reuse. This effect was confirmed by fewer misses for array *Y* with the new cache design—176—than with the standard cache shown in Figure 5b—480.

Locality analysis

The locality properties of numerical codes are relatively well understood because source code static analysis already provides a lot of information on intrinsic locality, and because the locality patterns are relatively simple (based on vector and matrix accesses). The locality properties of other types of code are less well understood. With CVT, we can examine a non-numerical code's cache behavior and derive inferences as to its locality characteristics.

For example, we collected a 100,000-entry trace by running the Unix command `compress` with Gordon Irlam's Spa software package (the 100-Kbyte target `compress` file contained ASCII text). This trace was extracted at a random point in the execution. Running this trace with the CVT showed that we actually extracted a heavily used code section because only 48 distinct program counters were repeatedly used. After running the trace, we obtained the cache miss layout of Figure 9.

Twenty percent of all misses occurred in a single cache line, and more than 10 program counters (load/store instructions) were using this particular cache line. However, only five were actually competing for it, breeding most cache misses for this line. Step-by-step execution showed that none of these instructions exploited spatial locality, so each access resulted in a miss. On the other hand, other references benefited from these accesses, since some references to this cache line were hits (the miss ratio of this line was roughly 30 percent).

CVT is dedicated to visualizing cache behavior of selected code sections rather than identifying critical code sections. We therefore intend to plug the CVT into a profiler similar to CProf that would address the first phase. Furthermore, by collecting information during the profiling run, such as loop boundaries and array subscripts' coefficients, we intend to reduce the number of references that need to be traced. CVT is freely available via FTP at ftp.prism.uvsq.fr in /pub/software/CVT. ❖

Acknowledgments

This work was supported by the European DGXIII BRA under Grant No. 6634. Elena Granston was supported by a postdoctoral research associateship in computational science and engineering under US National Science Foundation Grant No. CDA-9310307, and by the Center for Research on Parallel Computation under Grant No. CCR-9120008. Some of this work was done while Granston was at Leiden University, where she was supported by the Esprit Agency XIII under Grant No. 6634 BRA III.

References

1. M. Martonosi, A. Gupta, and T. Anderson, "Memspy: Analyzing Memory System Bottlenecks in Programs," *Performance Evaluation Rev.*, June 1992.
2. A.J. Goldberg and J. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with Mtool," *Proc. Supercomputing '91*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 481-490.
3. K. Kennedy, D. Callahan, and A. Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies," in *Instrumentation for Visualization*, ACM Press, New York, 1990.
4. A.R. Lebeck and D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, Oct. 1994, pp. 15-26.
5. M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPlan '91 Conf. Programming Language Design and Implementation*, ACM Press, New York, 1991, pp. 30-44.
6. M. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance of Blocked Algorithms," *Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1991.
7. S. Coleman and K.S. McKinley, "The Tile Size Selection Using Cache Organization and Data Layout," *ACM SIGPlan '95 Conf. Programming Language Design and Implementation*, ACM Press, New York, 1995.
8. K.S. McKinley and O. Temam, "A Quantitative Analysis of Loop Nest Locality," *Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996.
9. O. Temam, C. Fricker, and W. Jalby, "Cache Interference Phenomena," *Proc. ACM SIGMetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1994.
10. G. Cybenko et al., "Supercomputing Performance Evaluation and the Perfect Benchmarks," *Proc. IEEE Supercomputing '90 Conf.*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 254-266.
11. Q. Yang and L.W. Yang, "A Novel Cache Design for Vector Processing," *Int'l Symp. Computer Architecture*, ACM Press, New York, 1992.

Eric van der Deijl received an MS in computer science in 1996 from Leiden University, The Netherlands. He and Gerco Kanbier made the CVT, which was the topic of his master's thesis.

Gerco Kanbier received an MS in computer science in 1996 from Leiden University, The Netherlands.

Olivier Temam is an assistant professor at Versailles University in France. His research interests are in memory and processor, architecture with a special emphasis on performance analysis of cache memories. He received a PhD in computer science from Rennes University, France, in 1993.

Elena D. Granston works for Hewlett-Packard in Cupertino, California. Her research interests are in compilers, parallel processing, and computer architecture, with an emphasis on optimizing communication on large-scale multiprocessors. She received a BS in math and computer science, and an MS and a PhD in computer science, all from the University of Illinois, Urbana-Champaign. As a graduate student, she received the "Best Student Paper Award: Software" at Supercomputing '91.

Contact Temam at Olivier.temam@prism.uvsg.fr.