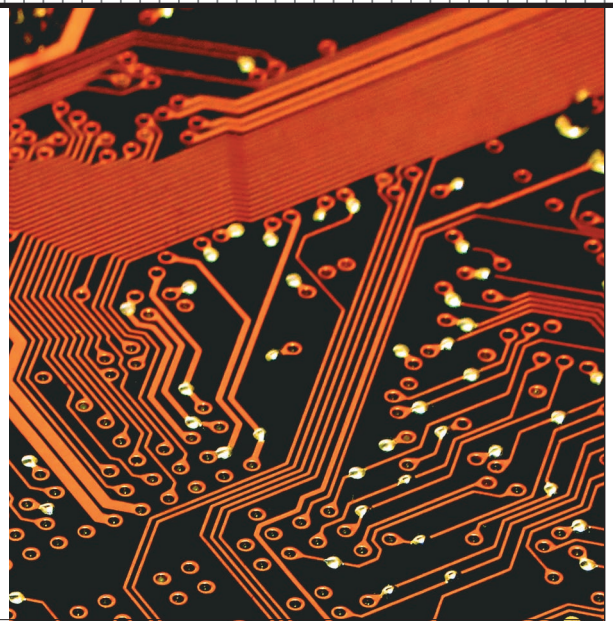


Refactoring for Data Locality

➔ Kristof Beyls, *Tele Atlas*

➔ Erik H. D'Hollander, *Ghent University*



Suggestions for locality optimizations (SLO), a cache profiling tool, analyzes runtime reuse paths to find the root causes of poor data locality, and suggests the most promising code optimizations. Refactoring using the hints of the SLO analyzer doubles the average execution speed of several SPEC2000 benchmark programs.

Refactoring a program means transforming its internal structure to improve its qualities, such as program organization, execution speed, or readability, without changing its functionality. Although refactoring is most often seen as a way to improve a program's internal architecture,¹ here we use the term to mean improving the execution speed. The main bottleneck often is not computation time, but rather memory access delay: Processors can execute hundreds of instructions in the time needed to fetch a single word from main memory.

A cache hierarchy narrows the performance gap between processor and memory. Only when the requested data is present in the cache does the system quickly deliver the data to the processor, saving it from data starvation. Basically, caches operate by retaining the most recently used data. If the processor reuses the data quickly, cache hits occur. Conversely, if it reuses the data after a long time, intervening data can evict the data from the cache, resulting in a *cache miss*. The majority of the processor chip area is typically reserved for caches. However, in many applications, cache misses cause the CPU to stall during more than half of the execution time. In these cases, execution speed benefits more from reducing the

number of cache misses than from reducing the number of computations.

Data accessed infrequently exhibits *low temporal data locality*. The corresponding cache miss arises from the instructions touching too much other data between use and reuse. The instruction trace that occurs between use and reuse of the same data is called the *reuse path*, and all code along the reuse path that accesses data contributes to the cache miss. Several cache profiling tools, such as Intel VTune,² Cprof,³ and Cachegrind,⁴ measure the hot spots where most cache misses occur and highlight the source code lines with the misses. Those highlighted lines, however, are merely the ends of the reuse paths that generate a cache miss. In many cases, refactoring other code along the reuse path improves the temporal data locality.

Figure 1 shows an example using our *reuse* profiling tool, called *suggestions for locality optimizations* (SLO; <http://slo.sourceforge.net>). The horizontal highlighted bars indicate the source code lines with cache misses, 95 percent of which occur in Line 5 of the function `inproduct`. Using cache profiling tools, the natural tendency is to rewrite `inproduct` for better cache performance. Unfortunately, refactoring of `inproduct` cannot diminish the data volume the processor accesses between

use and reuse. In fact, improving the locality and removing the resulting cache misses requires two different refactorings in function f . SLO indicates these refactorings as vertical bars to the left of the source code. Improving the locality of the remaining 5 percent misses at Line 29 requires a third refactoring.

In general, existing profilers pinpoint the cache miss location, but not the location that needs refactoring. Finding that location while knowing only the cache miss location is difficult in large programs. Compared to other tools,²⁻⁴ SLO is unique in that it precisely highlights the code regions where refactoring is needed to eliminate most cache misses, as in the Figure 1 example. SLO also indicates the type of refactoring needed, such as “FUSE” or “TILE” in the vertical bars. SLO’s automatic analysis enables programmers to improve their complex applications’ data locality.

In addition to the automatic data locality optimizations compilers perform, implementing the refactorings suggested by SLO improved speedups, with programs running 1.09 to 4.11 times faster.

LONG-DISTANCE REUSES CAUSE CACHE MISSES

Every cache hit originates from the reuse of data. However, data reuse results in a hit only if the data remains in the cache between use and reuse. We measure the temporal locality of a particular data reuse in terms of *reuse distance*. The reuse distance is the number of distinct data elements the processor accesses between the use and reuse of the same data.⁵ A reuse distance larger than the cache size generates a cache miss because the cache’s capacity is too small to store all data accessed between use and reuse. A reuse distance smaller than the cache size yields a hit—that is, for most cache policies.

Because reuse distance is independent of cache size, we can use it for different cache constellations; therefore, the reuse distance is a good general measure of a particular program execution’s data locality. Reuse distances smaller than the cache size exhibit good data locality, whereas longer distances cause misses, therefore exhibiting poor data locality.

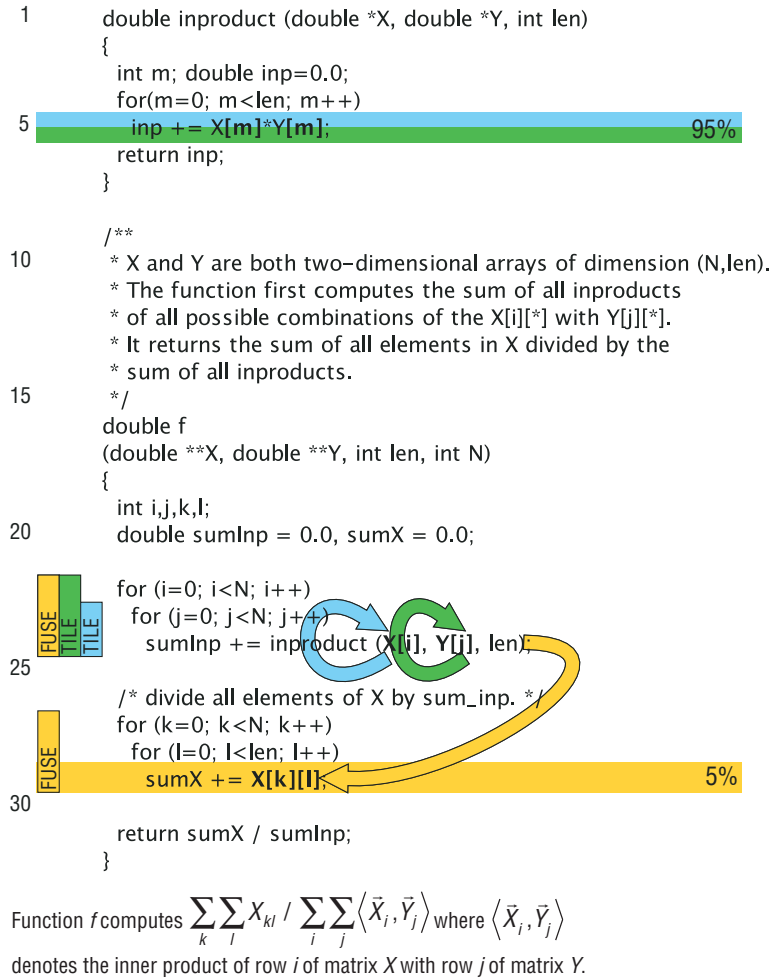


Figure 1. Suggestions for locality optimizations (SLO) visualization. SLO highlights source code lines containing cache misses with horizontal bars and suggests refactorings with vertical bars. The colored arrows indicate the main use-reuse pairs that have low temporal data locality and thus generate most cache misses.

CODE RESPONSIBLE FOR LONG-DISTANCE REUSE

Limiting the reuse distances to a value less than the cache size can eliminate cache misses. Typically, the distance is many times greater than the cache size. Also, the reuse path often spans large areas in the source code. All the code on the reuse path accessing data enlarges the reuse distance. Finding an effective code refactoring is complicated for two reasons: The reuse distance often must decrease by orders of magnitude, and the reuse path typically covers large code sections. Nonetheless, in many cases, a small refactoring of the appropriate source code fragment can reduce reuse distances to values below the cache size.

SLO manages the complexity of tracking reuse paths by building a *loop call tree*. The loop call tree is a new

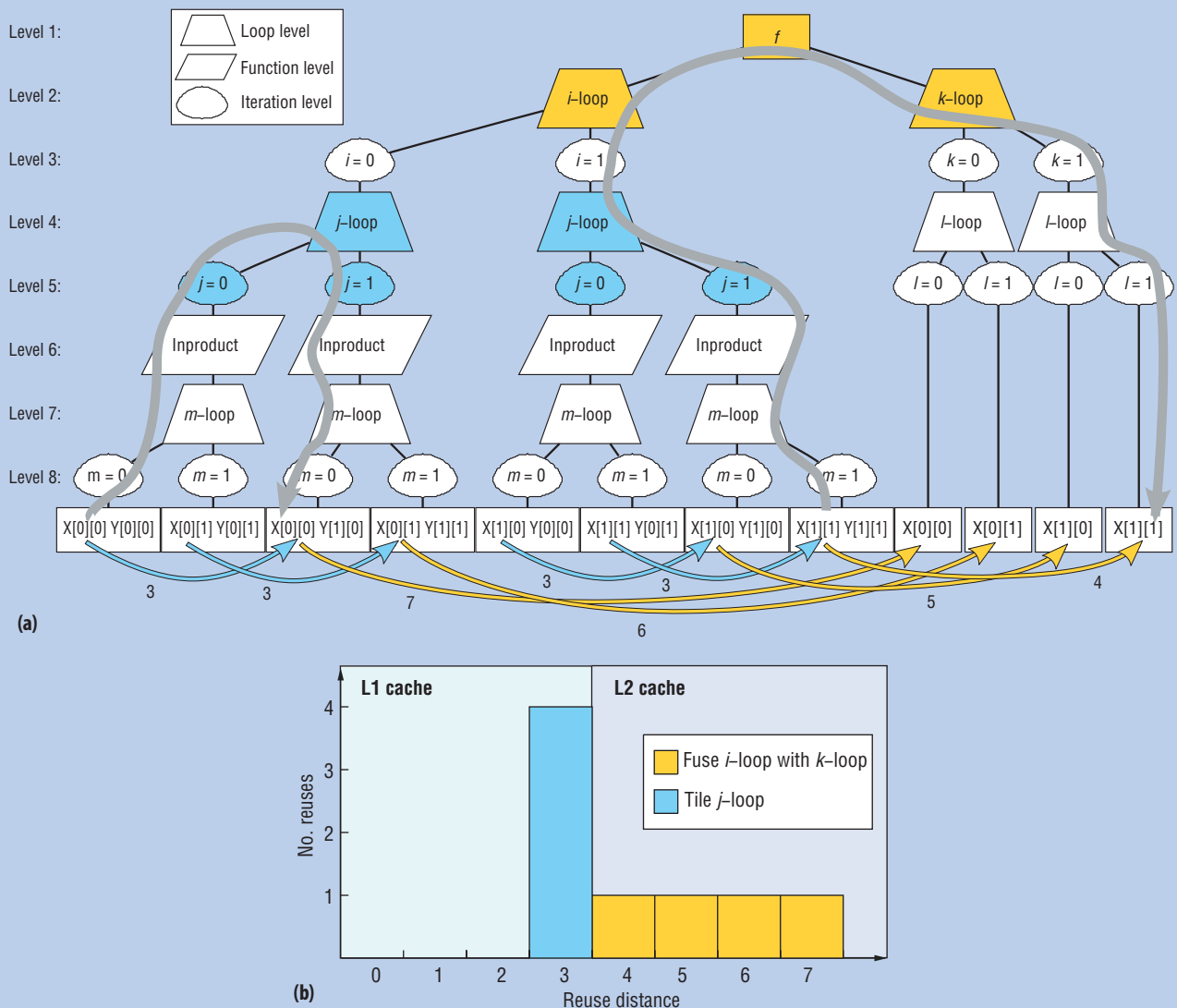


Figure 2. SLO's internal representation of reuses. SLO tracks reuse paths by building loop call trees, for example, in (a) the tree for the code in Figure 1 (for $N = 2$ and $\text{len} = 2$). The boxes at the bottom represent memory accesses in basic blocks, and the arrows join the uses with the reuses of *X*. The gray lines indicate reuse paths. (b) The distance histogram for the reuses (blue and yellow arrows) in 2a shows color-coded suggestions for refactorings. The bar colors correspond to the highlighted source code in Figure 1. The background colors indicate the distance ranges that result in hits in the L1 and L2 caches.

hierarchical representation of the program that contains a node for every function call, loop execution, and loop iteration observed at runtime. For example, Figure 2a shows the loop call tree for the program in Figure 1. The nodes near the top encompass large sections of the program execution.

The goal is to improve data locality by applying a code transformation to the reuse path, bringing use and reuse closer together. The most appropriate source code fragment needs to “see” all code that accesses data between use and reuse. Finding such a fragment is possible by inspecting the loop call tree.

The one node that sees the whole reuse path is the single node at the highest level of the reuse path—that is, the use and reuse nodes' least common ancestor. This is called the *overview node* because it provides a global view of the code executed between use and reuse. For example, in Figure 2a, the *j*-loop at Level 4 is the overview node of the leftmost reuse path. The use occurs in iteration *j* = 0, and the reuse occurs in iteration *j* = 1.

SLO uses vertical bars to highlight the source code corresponding to the overview node, as in Figure 1. The two blue overview nodes in Figure 2a correspond to the blue vertical bar in Figure 1. The blue bar on Line 5 indicates

the cache miss area. In this case, the j -loop generated half of the cache misses. In the loop call tree, the cache misses are at the end of the reuse paths.

The left reuse path makes clear why no refactoring in `inproduct` alone can eliminate the blue misses: The use and reuse occur in different calls to `inproduct` at Level 6. Therefore, when refactoring `inproduct`, this function sees only either the use or the reuse, but not both. Hence, a refactoring is necessary at a higher level in the loop call tree.

In addition to using vertical bars to highlight the refactoring region, SLO gives feedback by drawing arrows in the function containing the overview node. These arrows go from the beginning to the end of the reuse path in the refactoring region. For example, for the blue region in Figure 1, the reuse paths accessing array elements X enter and exit the region through the call to `inproduct`. For the yellow region, the reuse paths start in the call to `inproduct` and end in expression $X[k]$.

Highlighting all source code lines on the reuse path, as the reuse distance visualizer (RDVIS) tool does,⁶ can also help find a refactoring. However, in cases where reuse paths span thousands of lines, finding a location in the reuse path where a small code change largely reduces the reuse distance is laborious. In our experience, highlighting only the region corresponding to the overview node helps to quickly understand the root cause of long reuse paths.

SHORTENING REUSE DISTANCE

SLO highlights the source locations of the largest possible reduction in reuse distance. Although the programmer is free to change the code in any way, SLO suggests a refactoring that merges the overview node's children nodes at runtime. Merging the children nodes provides more freedom to reschedule the uses and reuses from each child node. The loop call tree has three types of nodes: *loop level*, *iteration level*, and *function level*. For each type, SLO suggests a different source code transformation that results in merged children nodes at runtime. The "Loop Transformations That Increase Locality" sidebar provides background on the code transformations.


With loop-level nodes, use occurs in one loop, and reuse occurs in another loop. Fusing both loops can draw these reuses nearer. The yellow nodes in Figure 2a represent an example of this case. Use occurs in the i -loop, and reuse occurs in the k -loop. SLO indicates this with the "FUSE" vertical bars.

In iteration-level nodes, use and reuse occur in different iterations of the same loop. It's possible that each iteration addresses more data than the processor can store in the cache. To draw the reuses closer together, each iteration should access less data. An example of an

iteration-level node is the reuse path crossing the blue iteration nodes $j = 0$ and $j = 1$ in Figure 2a. Two traditional transformations that reduce the amount of data accessed in a single iteration are loop interchange and loop tiling. SLO indicates this case with the "TILE" vertical bar.

With function-level nodes, use occurs in one function, and reuse occurs in another function. The solution comprises two steps. First, we put the bodies of the two functions in a common function. Next, we fuse the code that produces the uses with the code that generates the reuses. SLO indicates this case with "FUSE" vertical bars next to the two function calls' source code.

In Figure 2b, yellow and blue indicate two required refactorings. We can shorten the yellow reuses by fusing the i -loop with the k -loop and the blue reuses by tiling the j -loop. In Figure 1, the green refactoring annotation suggests tiling the i -loop to avoid cache misses resulting from the reuses of array Y .



With function-level nodes,
use occurs in one function,
and reuse occurs in
another function.

USING SLO TO OPTIMIZE PROGRAMS

To implement measuring the reuse distances and the loop call tree, we extended the GNU Compiler Collection (GCC) to instrument the memory accesses, function calls, and loops. The resulting instrumented code is linked with a library that processes the memory access, loop execution, and function call events at runtime to track the data reuses.

For each reuse, the instrumentation library calculates the reuse distance and the children of the overview node in the loop call tree. Further, it calculates the basic blocks corresponding to the overview node's children. SLO maps those basic blocks to the corresponding pair of source code locations (such as loop or function call site). For each pair of source locations, the library records the histogram of reuse distances. When the program finishes, it writes to a file the mapping of source location pairs to the corresponding reuse distance histogram.

To speed up this measurement, we use advanced sampling techniques.⁷ Two modes of measurement are available. The first mode measures the reuse distance, and the second mode measures the reuse data volume—that is, the total number of accesses between reuses. The instrumented program runs about 100 times slower when measuring the reuse distance and five times slower when measuring the reuse data volume. Whereas we can

LOOP TRANSFORMATIONS THAT INCREASE LOCALITY

Loops iterating over large data structures are prone to poor data locality. Reordering the iterations to shorten long reuse distances can improve locality. Three well-known loop transformations improve memory-access behavior: loop fusion, loop interchange, and loop tiling.¹

When the processor traverses the same data twice in different loops, it's usually beneficial to merge, or fuse, the computations into a single loop to iterate over the data structure only once.

Figure A1 shows the source code, memory access stream, and reuse distance histogram before and after applying loop fusion. The fusion reduces the reuse distances from $M - 1$ to 0.

When the processor reuses the same data between different iterations of an outer loop, it can reduce the gap between the reuses by interchanging the outer and the inner loop. As a result of loop interchange, the iterations reusing the same data execute near each other. In the example in Figure A2, the reuse distances decrease from $N - 1$ to 0.

When the processor reuses some data in an inner loop and other data in an outer loop, loop interchange will optimize only one of the data reuses. The other reuse remains between the outer loop's iterations. Loop tiling improves both types of reuses at the same time. This is done by limiting the amount of data the processor accesses in the inner loop. Blocks of a limited number of iterations are called *tiles*.

As a result of loop tiling in the Figure A3 example, the reuses of $B[j]$ in the second loop are 10 iterations apart. This gives a constant reuse distance of 11, as opposed to the reuse distance $N + 1$ before tiling.

After tiling, the reuse distance remains constant with growing problem size N . Furthermore, the i -loop interrupts the reuses of $A[i]$ between iterations of the j -loop only every 10 iterations. Therefore, only one-tenth of those reuses grows large—that is, $M + 19$. With a simple loop interchange, all those reuses would grow large.

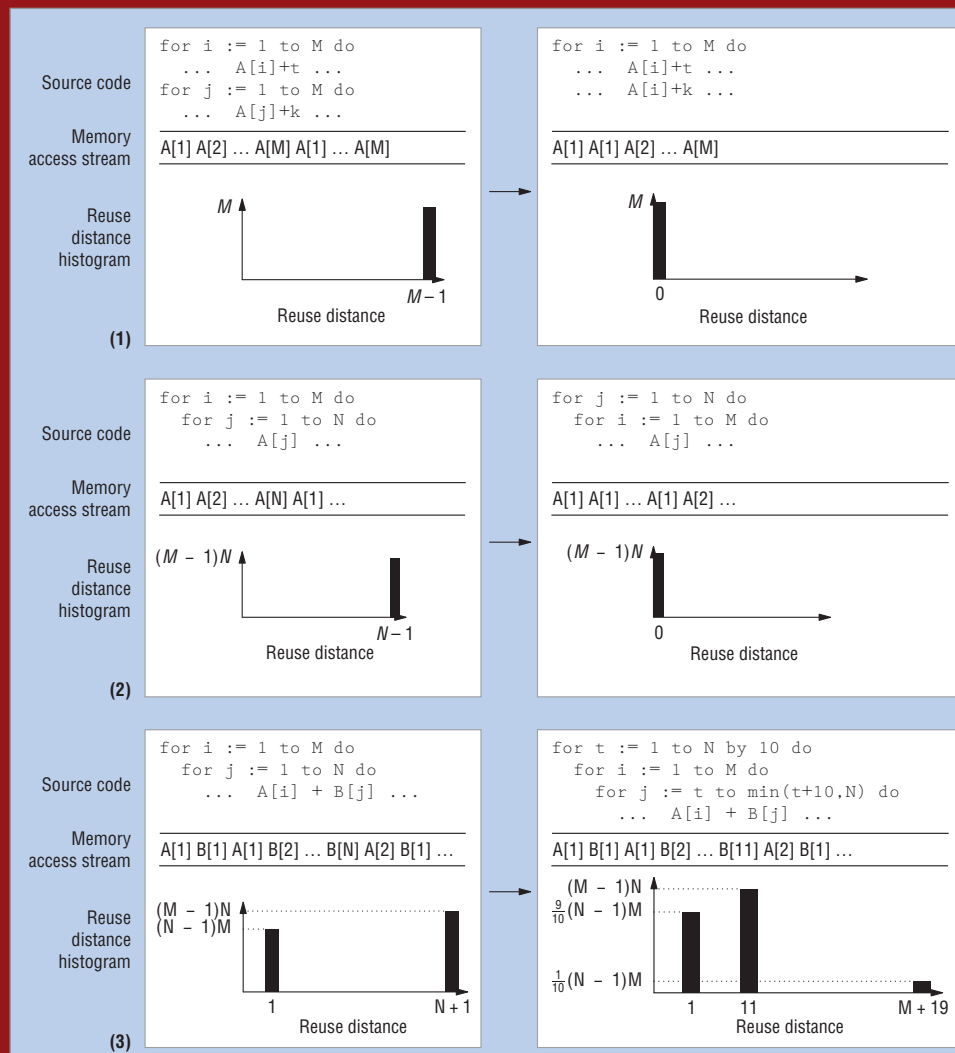


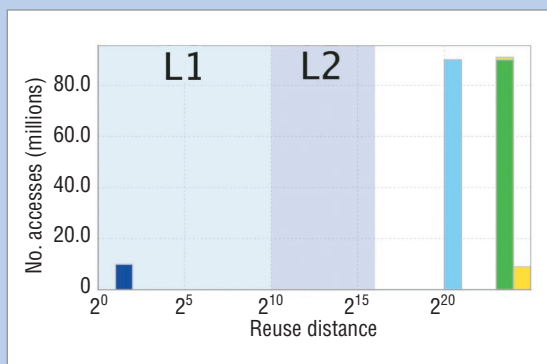
Figure A. Loop transformations. Three loop transformations can improve memory-access behavior: (1) loop fusion, (2) loop interchange, and (3) loop tiling. These examples show the source code, memory access stream, and reuse distance histogram before (left column) and after (right column) applying the transformations.

Reference

1. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, 2002.

use the second mode to quickly discover the most promising areas for refactoring, the first mode gives complete reuse distance information useful for estimating cache behavior.

After the profiling run, SLO reads the recorded reuse distance histograms and individually colors and stacks them, as in Figure 3a. The horizontal reuse distance axis is scaled logarithmically. When the user clicks a colored bar



(a)

```
double f
(double **X, double **Y, int len, int N)
{
    int i,j,l;
    double sumInp = 0.0, sumX = 0.0;

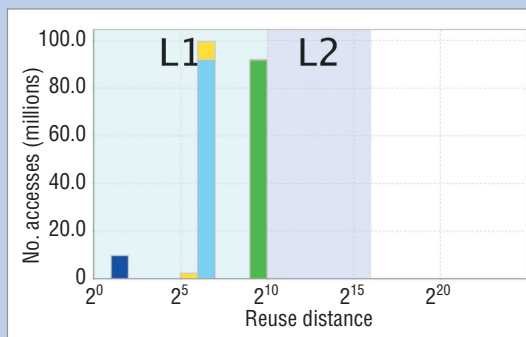
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++)
            sumInp += inproduct (X[i], Y[j], len);
        for (l=0; l<len; l++)
            sumX += X[i][l];
    }
    return sumX / sumInp;
}
```

(b)

```
double f
(double **X, double **Y, int len, int N)
{
    int i,j,l;
    double sumInp = 0.0, sumX = 0.0;
    int T;
    /* loop over X and Y in tiles of maximum 50
       elements. T points to the beginning of
       the current tile. */
    for (T=0; T<len; T += 50) {
        /* Tsize is the size of the tile: either 50,
           or possibly less in the last tile. */
        const int Tsize = (T+50>len) ? len-T : 50;
        for (i=0; i<N; i++) {
            for (j=0; j<N; j++)
                sumInp +=
                    inproduct (&X[i][T], &Y[j][T], Tsize);
            for (l=T; l<T+Tsize; l++)
                sumX += X[i][l];
        } /* end of i-loop. */
    } /* end of T-loop. */

    return sumX / sumInp;
}
```

(c)



(d)

Figure 3. Implementing SLO's suggestions. (a) SLO suggests three different refactorings for reuses at a distance greater than the L2 cache size. These refactoring colors correspond to Figure 1: tiling the *i*-loop and the *j*-loop (blue and green, respectively) and fusing *i*-loop with *k*-loop (yellow). The refactoring produces code after (b) loop fusion and (c) loop tiling. (d) A reuse histogram shows that reuse distances are decreased after optimization compared to 3(a). The background color shows that the L2 cache-missing reuses have been turned into L1 cache-hitting reuses after applying the refactorings.

in the histogram, SLO highlights the corresponding code, akin to the bars and arrows in Figure 1.

The three rightmost bars in the histogram in Figure 3a indicate that SLO suggests three refactorings. First, the yellow bars in the Figure 1 source code indicate fusing the *i*-loop and *k*-loop. The yellow arrow indicates that the

uses occur inside the call to `inproduct` on Line 24, while reference `X[k]` generates the reuses. The *i*-loop computes `sumInp`, and the *k*-loop computes `sumX`. The computations in both loops are completely independent: The first loop doesn't use `sumX`, and the second loop doesn't use `sumInp`. Therefore, we can safely fuse both loops (see Figure 3b).

Next, the green and blue bars in Figure 1 suggest tiling both the i -loop and the j -loop. The processor simply accesses too much data in a single iteration of the inner j -loop and the outer i -loop. Now, the green and blue arrows reveal that the reuses occur between different calls to `inproduct`. Indeed, a single iteration of the j -loop, which computes the inner product $\langle X[i], Y[j] \rangle$, accesses 2×10^6 elements, as arrays $X[i]$ and $Y[j]$ each contain 10^6 elements. The refactored code in Figure 3c, following SLO's tiling suggestions, accesses fewer elements in a single iteration of the i -loop and j -loop. Rather than computing the inner products of long arrays with length `len`, the refactored code calculates the inner product of a subvector with length `Tsize = 50` elements in each iteration. This method limits the number of data elements accessed in a single iteration to 100, down from 10^6 .

The rewards of improved data locality and diminished execution time often offset increased code complexity.

As Figure 3d shows, as a result of the two refactorings, the reuse distance decreased by several orders of magnitude. The cache-missing reuses at distance 2^{20} to 2^{25} have transformed into L1 cache-hitting reuses at distance 2^5 to 2^{10} . After refactoring, the program runs about two times faster on a Pentium 4 PC.

This example also shows that refactoring for data locality might increase the code complexity. However, the complexity increase is mitigated in two ways. First, the code refactoring is localized to the function or loops the overview node selects. Second, the suggested refactorings are well-known structured transformations. The rewards of improved data locality and diminished execution time often offset increased code complexity. Essentially, using SLO involves two steps: The programmer finds the largest contribution in the histogram to the right of the cache size (yielding the most cache misses) and then analyzes the corresponding code and implements SLO's suggestions.

SPEC2000 CASE STUDIES

The industry often uses the SPEC2000 benchmark suite to measure a computer's CPU and memory performance.⁸ We used SLO to analyze seven SPEC2000 benchmark programs for data locality.

Locality patterns

Figure 4 shows the generated locality patterns. We started by inspecting the area in the histograms to the

right of the L2 cache size because this corresponds to the reuses that generate the most costly L2 cache misses.

Each color represents a different refactoring suggestion. Clicking a color highlights the refactoring region in the code. For example, for *Applu*, Figure 4 shows the refactoring region corresponding to the blue bar (Fortran 77 code). It indicates that decreasing the blue long-distance reuses requires fusing functions `jac1d` and `blts`. Further investigation of the source code reveals that both `jac1d` and `blts` consist of a single large loop nest with the same loop bounds. Inlining both functions and fusing the two loops would easily shrink the reuse distances.

We analyzed the other programs in the same way and obtained the following results.

Applu solves coupled, nonlinear partial-differential equations. We can remove the blue, green, yellow, and orange bars at distances greater than 2^{20} by following SLO's suggestion to fuse several functions. The other bars remain because either they're small and therefore less important, or we couldn't find a legal transformation.

Galgel computes the convective motion in water that differences in temperature generate. The bars on the right correspond to loops with long-distance reuses between iterations. We shortened the distances by reducing the size of the arrays traversed.

Art recognizes objects in an image by simulating neural networks. The red peak at distance 2^{17} corresponds to an outer loop that accesses too much data in each iteration. We reduced the data volume by recomputing intermediate data values between iterations, instead of storing them in memory. The colors in the peak at distance 2^{14} represent pairs of loops that need to be fused. We also applied these fusions.

Crafty is a chess program. The histogram shows that the data locality is quite good: Most of the reuses hit in the L1 cache. SLO highlights the few L1 misses in a single phase (the red area in the histogram). This phase corresponds to the loop that iterates over the evaluation of the legal moves at a given board position. Eliminating the remaining small fraction of L1 misses will require further investigation of this code.

GCC compiles C codes to an 88 K assembler. The histogram shows that many refactorings are required to improve the locality, and the program has few hot spots. An exception is the red peak at distance 2^{22} , corresponding to the convergence loop that calculates liveness for register allocation. Alas, tiling this loop would result in more convergence iterations, resulting in an overall slowdown.

Versatile Place and Route places and routes electronic circuits for field-programmable gate arrays. The red, green, and blue refactorings require tiling the loops in Dijkstra's shortest path algorithm, which can't be done because of dependencies. We removed the orange long-distance reuses, however, by using a more efficient memory allocator.

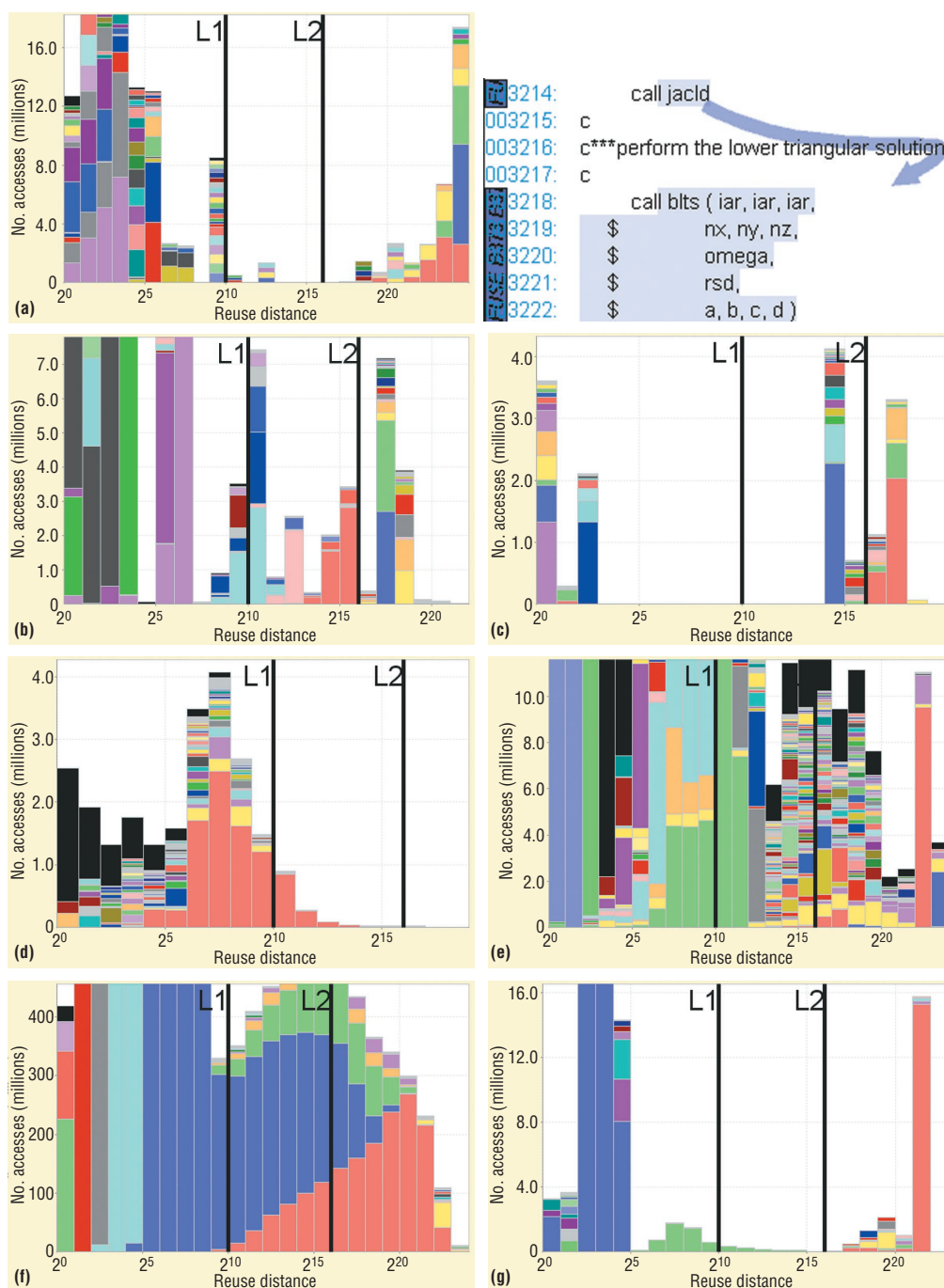


Figure 4. SLO-generated locality patterns. The analysis included seven SPEC2000 benchmark programs: (a) Applu, (b) Galgel, (c) Art, (d) Crafty, (e) Gnu Compiler Collection, (f) Versatile Place and Route, and (g) Equake. Clicking on a color highlights the refactoring region in the code, as in the Applu example.

Equake simulates earthquakes. A sparse matrix represents a 3D model of the simulated region. The red peak at distance 2^{21} corresponds to iterating over the large

sparse matrix between time steps in the simulation. We tiled the corresponding loop.

In the example program in Figure 3, three suggested

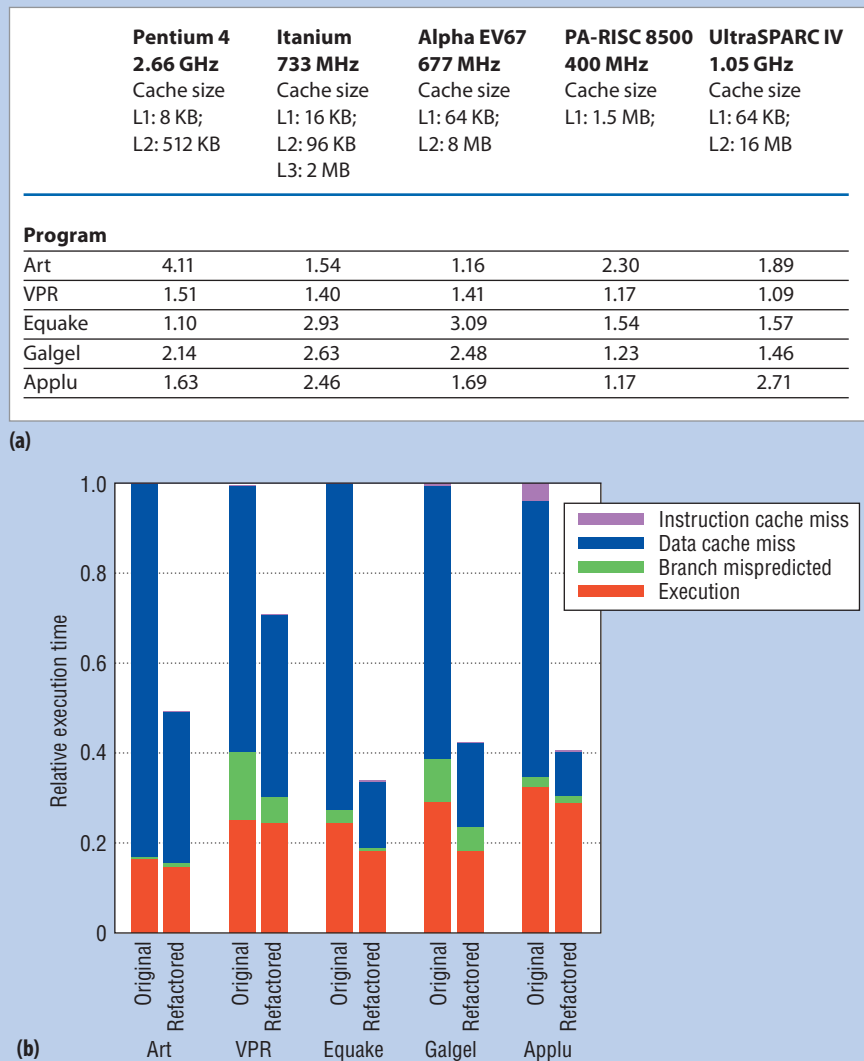


Figure 5. Data cache stall time reduction and speedup after refactoring. (a) The table shows speedups resulting from SLO's suggested refactorings on five platforms. (b) The chart shows a breakdown of execution time on an Itanium processor for original and refactored codes.

refactorings eliminate all L2 cache misses. For all but one of the analyzed SPEC2000 programs, SLO indicates that 11 or fewer refactorings are needed to optimize 90 percent of the L2 misses. The exception is GCC, where the numerous compiler phases result in fewer focused hot spots.

We optimized five of the seven programs, following the suggested refactorings. We chose not to optimize Crafty and GCC because SLO indicates that Crafty has no significant locality problems and GCC lacks hot spots. For the other five programs, temporal locality improved significantly. Similar to the example in Figure 3, the refactorings move the reuse distance peaks in the histogram to the left. In general, refactoring necessitates a good understanding of the code to facilitate

analyzing the code and data flow and verifying that the planned refactorings are legal. On average, the actual rewriting and refactoring of the code proved to be the easiest part of the process; on average, it took one to two hours. For the case studies, the analysis of SLO's suggestions, understanding what the analyzed code does exactly, checking the legality of the suggested transformations, and the actual code refactoring took an average of two working days per program.

Execution time

Figure 5a shows the resulting speedups on five platforms. Figure 5b shows a detailed breakdown of the execution time on an Itanium processor, as measured by performance counters. Refactoring eliminated a large fraction of the stall time from data cache misses. The original and refactored source codes were compiled with all optimizations enabled. In principle, the SLO suggestions could help a compiler to find better data locality optimizations. However, the long reuse paths, dynamic data structures, and complex control flow often prevent the compiler from concluding that a refactoring is legal. This is why SLO needs a

human programmer in the loop. A programmer can accurately determine whether a refactoring is legal on the basis of the algorithm and the programmer's knowledge of the field and ability to adjust the code to make a refactoring possible. This process is why refactoring results in speedups ranging from a factor of 1.09 to 4.11.

The speedup varies across platforms owing to microarchitectural differences that affect the slowdown resulting from cache misses. Significant characteristics are the sizes and access latencies of the cache levels, out-of-order execution versus in-order execution, and the prefetch mechanisms. Nonetheless, all speedups are positive, showing that improving the temporal locality has a positive impact on all platforms and programs. The

source code of the five refactored programs is available at the SLO website (<http://slo.sourceforge.net>).

Eliminating cache misses

Cache misses occur when too much data is accessed between use and reuse—that is, when the reuse distance is larger than the cache size. To turn misses into hits, the reuse distance must be reduced to a value smaller than the cache size. All the code on the execution path between use and reuse contributes to enlarging the distance.

Traditional profilers highlight the reuse path's end point where the miss occurs. However, eliminating the cache miss often requires refactoring a completely different source location on the reuse path. Therefore, SLO highlights a small code region on the reuse path where refactoring has the highest impact on reuse distance. The region is determined based on the principle that the code with an overview of both the use and the reuse has the largest impact on the reuse distance. This is found by looking up the highest level in the hierarchy of function calls and loops traversed by the reuse path.

Evaluation of SLO using SPEC2000 programs covering several important application domains revealed that the execution speed was doubled on the average on five different platforms. This indicates that SLO improves the locality in a platform-independent way, resulting in significant speedups.

Refactoring for data locality opens a new avenue for performance-oriented program rewriting. SLO has broken down a large part of the complexity that software developers face when speeding up programs with numerous cache misses. Therefore, we consider SLO to belong to a new generation of program analyzers. Whereas existing cache profilers (generation 1.0) highlight problems such as cache misses, second-generation analyzers (such as SLO) highlight the place to fix problems.

Improving data locality is also important in hardware-based applications. SLO was already used to optimize the frame rate and energy consumption in a wavelet decoder implemented on an FPGA.

In another vein, the SLO concepts could be incorporated in interactive performance debuggers and

profile-directed compilers. We believe that SLO will be useful in the optimization of many data-intensive applications. ■

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
2. M. Atkins and R. Subramaniam, "PC Software Performance Tuning," *Computer*, Aug. 1996, pp. 47-54.
3. A.R. Lebeck and D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, Oct. 1994, pp. 15-26.
4. N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science*, Oct. 2003, pp. 1-23.
5. C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse Distance Analysis," *Proc. Conf. Programming Language Design and Implementation*, ACM Press, 2003, pp. 245-257.
6. K. Beyls, E.H. D'Hollander, and F. Vandeputte, "RDVIS: A Tool That Visualizes the Causes of Low Locality and Hints at Program Optimizations," *Proc. Int'l Conf. Computational Science*, LNCS 3515, Springer, 2005, pp. 166-173.
7. K. Beyls and E.H. D'Hollander, "Discovery of Locality-Improving Refactorings by Reuse Path Analysis," *Proc. Int'l Conf. High Performance Computing and Comm.*, LNCS 4208, Springer, 2006, pp. 220-229.
8. J.L. Henning, "SPEC CPU2000: Measuring CPU Performance in a New Millennium," *Computer*, July 2000, pp. 28-35.

Kristof Beyls is a software engineer at Tele Atlas. His research interests include program analysis, compilers, high-performance computing, and software engineering. Beyls received a PhD in computer science from Ghent University, Belgium. Contact him at kristof.beyls@elis.ugent.be.

Erik H. D'Hollander is a professor in the Department of Electronics and Information Systems at Ghent University, Belgium. His research interests include parallel computing, high-performance architectures, compilers, and embedded systems. D'Hollander received a PhD in computer science from Ghent University. Contact him at erik.dhollander@elis.ugent.be.

For more information on any topic presented in *Computer*,
visit the IEEE Computer Society Digital Library at

www.computer.org/csdl