Journal of Instruction-Level Parallelism 10 (2008) 1-9 Submitted 10/06; published 6/08

Refactoring Intermediately Executed Code to Reduce Cache Capacity Misses

Kristof Beyls KRISTOF.BEYLS@ELIS.UGENT.BE
Erik H. D'Hollander ERIK.DHOLLANDER@ELIS.UGENT.BE
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41
B-9000, Ghent, Belgium

Abstract

The growing memory wall requires that more attention is given to the data cache behavior of programs. In this paper, attention is given to the capacity misses i.e. the misses that occur because the cache size is smaller than the data footprint between the use and the reuse of the same data. The data footprint is measured with the reuse distance metric, by counting the distinct memory locations accessed between use and reuse. For reuse distances larger than the cache size, the associated code needs to be refactored in a way that reduces the reuse distance to below the cache size so that the capacity misses are eliminated.

In a number of simple loops, the reuse distance can be calculated analytically. However, in most cases profiling is needed to pinpoint the areas where the program needs to be transformed for better data locality. This is achieved by the reuse distance visualizer, RDVIS, which shows the intermediately executed code for critical data reuses. In addition, another tool, SLO, annotates the source program with suggestions for locality optimization. Both tools have been used to analyze and to refactor a number of SPEC2000 benchmark programs with very positive results.

1. Introduction

Moore's Law has been relatively robust in predicting a doubling of the speedup of a processor every 18 months. This corresponds to a speedup increase of 59% per year. In contrast, the speedup of the main memory advances only about 7% per year. As a consequence, the speed gap between processor and main memory doubles every 21 months. Instruction and data caches have been used to bridge this gap for a long time, and their performance depends on the hit rate. Cache misses have been classified as cold misses, occurring the first time data enters into the cache; conflict misses, which occur when two memory addresses are mapped onto the same cache line and capacity misses, which occur when the cache size is too small to contain all the data addressed between the use and the reuse of the same memory location. Cold misses are hardly avoidable and conflict misses have been successfully reduced by architectural innovations and compiler optimizations. Capacity misses on the other hand are caused by memory accesses widely separated in place and time, which makes it difficult to trace their occurrence. In this paper, the nature of the capacity misses is related to the distance between the use and the reuse of the same memory location. In the first part, the reuse distance metric is presented, and it is shown that this metric can be calculated analytically in function of the loop parameters for simple loops. In the second part, a method is given to dynamically measure the reuse distance of a program in execution and to find the code which is responsible for long reuse distances. In the third part, an interprocedural analysis is presented, which allows to automatically annotate the program with

This article initially appeared in abbreviated form in ACM Computing Frontiers 2006.

refactoring hints to obtain a better data locality. In the last part, the techniques presented are used to refactor a number of SPEC benchmarks for better data locality. As a result, these benchmarks run significantly faster on different platforms.

2. Reuse Distance Equations

In order to reduce the number of capacity misses, it should be understood what causes them. Most regular reuse occurs in loops and loops lend themselves for an analytical treatment. Loops are also most easily transformed to change the data access patterns and granularity. Consider a memory access stream, identified by a series of read/writes to different addresses. The reuse distance is the number of unique locations between the two references with the same address. E.g., consider the stream in Figure 1.

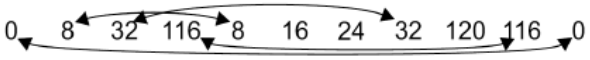


Figure 1: Memory References and Their Reuse Distances. The reuses distance of locations 8, 32, 116 and 0 are respectively 2, 4, 5 and 6.

The iteration space of a nested loop can be described by set of inequalities in the loop indices. The volume of integer indices so obtained is a polytope (a bounded polyhedron). E.g., consider two iteration points I_r and I_s respectively in the first and second loop of the following two loops:

```
for i := 1 to N
  for j := 1 to i
    A(i,j) := ...
  endfor
endfor
for k := 1 to N
  for l := 1 to k
    A(k,l) := ...
  endfor
endfor
```

The locations touched between the use of **A(i,j)** in the first loop and the reuse as **A(k,l)** in the second loop are subject to the following conditions (Figure 2):

1. (i,j) must belong to iteration space of the first loop,
2. (k,l) must belong to the iteration space of the second loop,
3. use and reuse address the same location, i.e., i = k and j = l,
4. the locations addressed after A(i,j) in the first loop are $\{(x,y) : (x=i \wedge i<y\leq N) \vee (I<x\leq N \wedge 1\leq y\leq x)\}$, and
5. the locations addressed before A(k,l) in the second loop are $\{(x,y) : (1\leq x<k \wedge 1\leq y\leq k) \vee (x=k \wedge 1\leq y< l)\}$.

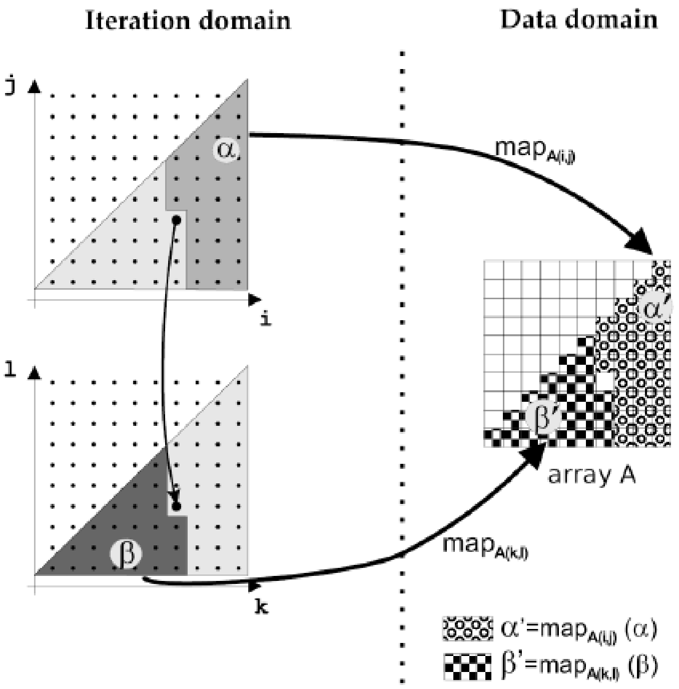


Figure 2: Iteration Space and Memory Referenced between Use and Reuse of **`A(i,j)`.**

Combining these conditions, and substituting k=i and l=j yields the following Presburger formula [1] for the addressed data set of reference **A(i,j)** between use and reuse:

$$\begin{aligned} \mathrm{ADS}(\mathtt{A}(\mathtt{i},\mathtt{j})) \; = \; & \{(x,y) : (x,y) \in \mathbb{Z}^2 \wedge \\ & 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge \\ & ((x = i \wedge i < y \leq N) \vee (i < x \leq N \wedge 1 \leq y \leq x) \vee \\ & (1 \leq x < i \wedge 1 \leq y \leq i) \vee (x = i \wedge 1 \leq y < j)) \; \} \end{aligned} \qquad (1)$$

The count of the integer solution points (x,y) subject to the linear constraints in (1) is the reuse distance. The count is parameterized in the loop bound *N* and can be expressed by an Ehrhart polynomial. The calculation of the Ehrhart polynomial is achieved either by using polylib [2] or using the method of Barvinok [3,4]. Unfortunately, the analytical calculation of the reuse distance is much more complicated or intractable for real programs.

The analytical calculation of the reuse distance may be embedded in a cache aware compiler. In that case, it is plausible that the compiler uses program transformations to reduce the reuse distance, for example by fusing both loops in the example given, subject to a dependence analysis. This intervention by the compiler is limited for a number of reasons. First, large reuse distances are typically caused by complex instruction paths, which impede analytical treatment. Second, a single pair of memory references in a program may entail reuses at many different memory locations, each with their own reuse distances. Third, there is no known analytical way to determine the instruction path between use and reuse. In the following sections a method is presented to consistently identify the data references in the program causing large reuse distances and associated capacity misses. In addition, data reuses with overlapping instruction paths are clustered. Improving the locality of a common instruction path will benefit all data reuses in the cluster. In parallel, regular structures such as loops are annotated with hints to improve the data locality. It is demonstrated that refactoring code segments based on these the hints substantially improve the locality and execution time of a number of SPEC benchmark programs.

In the next section, a method is described to measure and order the reuse distances of a program execution. Then, a basic block vector is introduced to mark the execution trace between the use and reuse of a particular memory location. Clustering similar basic block vectors, identifies parts of the program with poor data locality. In parallel with these calculations, a graphical environment is introduced, which allows easy navigation to various parts of the program susceptible for better data locality after refactoring.

3. The Reuse Distance of Real Programs

In this section, basic terms and definitions are introduced to characterize reuses in a program.

Definition 1: *A memory access a_x is a single access to memory that accesses address x. A memory reference r is the source code construct, e.g.,* `A(i,j)` *that generates a memory read or write instruction at compile-time, which, in turn, generates a memory access at runtime. A memory access trace T is a sequence of memory accesses, indexed by a logical time. The difference in time between consecutive accesses in a trace is 1. The time of an access a_x is denoted by $T[a_x]$.* □

Definition 2: *A* ***reuse pair*** (a_x, a'_x) *is a pair of memory accesses in a trace such that both accesses address the same data, and there are no intervening accesses to that data. The use of a reuse pair is the first access in the pair; the reuse is the second access. A* ***reference pair*** **`(r1,r2)`** *is a pair of memory references. The reuse pairs associated with a reference pair* **`(r1,r2)`** *is the set of reuse pairs for which the use is generated by* **`r1`** *and the reuse is generated by* **`r2`***, and is denoted by* ***reuses*** **`(r1,r2)`**. □

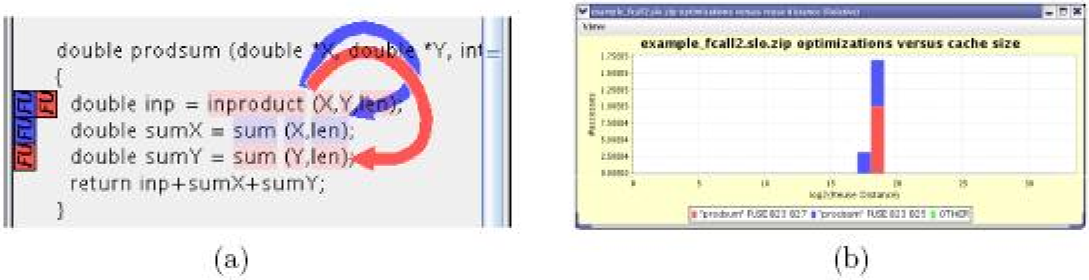
Definition 3: *The* ***Intermediately Executed Code*** *(IEC) of a reuse pair (a_x, a'_x) is the code executed between $T[a_x]$ and $T[a'_x]$.* □

Definition 4: *The* ***reuse distance*** *of a reuse pair in a trace is the number of unique memory addresses in that trace between use and reuse.* □

In essence, in the source code, only reference pairs are visible (e.g., **A(i,j)** → **A(k,l)**). When these reference pairs access equal memory locations *x*, a reuse pair (a_x, a'_x) is associated with it. Both accesses a_x and a'_x differ by the time (or sequence number) they occur in the memory trace. Typically a reference pair generates many reuse pairs. Cache misses are identified by the reuses that have a distance larger than the cache size [5].

Since each reference pair covers many reuse pairs, each with its own reuse distance, the reuse distances of a reference pair are represented by a histogram. E.g., in Figure 3(b), the reuse histograms of two reference pairs are plotted with different colors. The corresponding reference pairs are indicated by arrows in the source code (see Figure 3(a)).

To turn a cache miss into a cache hit, the reuse distance of the corresponding reuse pair must be reduced to be smaller than the cache size [5]. To find a code transformation that reduces the reuse distance of a particular reuse pair, however, it is not enough to know the source and the sink of a reuse pair. It is also necessary to know the code trajectory causing the long-distance cache misses. This *reuse path* is called the intermediately executed code (or IEC) and a method to track this code is presented next.



```
double prodsum (double *X, double *Y, int
{
  double inp = inproduct (X,Y,len);
  double sumX = sum (X,len);
  double sumY = sum (Y,len);
  return inp+sumX+sumY;
}
```

Figure 3: Reference pairs indicated by arrows and their corresponding reuse histogram.

4. Identifying and Clustering Reuse Paths by Basic Block Vectors

In order to identify program traces with the proper granularity, a basic block is chosen. The basic blocks of the program are numbered consecutively, and a trace in the program between use and reuse is indicated by the set of basic blocks which are executed. For a *single* trace, the basic blocks are represented by an *n*-bit vector of zeros and ones, where the ones indicate the basic blocks executed. Of course, many different traces may exist, each joining some uses and reuses of a reference pair. To accommodate for *multiple* traces, the executed blocks are weighted against the total number of traces to obtain their relative execution frequency in the path between use and reuse. As a result, each reference pair has an associated basic block vector [6] containing the relative execution frequencies of each basic block. This extension is still compatible with the definition of a basic block vector for a single trace.

Cache misses are reduced by transforming or refactoring the code along reuse paths to shorten many reuse distances. It is therefore useful to find reuse paths, which are common to a large number of reuse pairs. Reuse paths affecting many reuses are obtained by comparing the basic block vectors of many reuse pairs and clustering the blocks belonging to similar basic block vectors. The algorithm uses nearest neighbor clustering, where the distance between adjacent vectors can be arbitrarily selected between zero and infinity. This allows the user to find an appropriate measure, which forms a cluster with many reuse pairs and maps onto tractable code segments for refactoring. This is achieved by the Reuse Distance Visualizer, RDVIS.

After finding a bundle of long reuse distances, the reuse distance visualizer highlights the intermediately execute code in the program. By the nature of a long reuse distance, this typically involves large areas of program, including nested loops and functions. A further analysis of the program structure permits to focus on the code which offers the highest potential for data locality improvement. The SLO tool suggests locality optimizations by finding the most promising loops and highlighting these loops with the code transformation hints.

5. Analyzing Control Flow Using the Least Common Ancestor Function

In general the use and the reuse of a reuse pair may occur in different loops and functions of the program. The function in which both the use and the reuse are visible is called the least common ancestor function or LCAF. The basic block in the LCAF where the use occurs is called the Use

Basic Block (UBB) and a basic block that contains the reuse is called the Reuse Basic Block (RBB). Note that the use and the reuse basic blocks may contain function calls in which the actual memory access of the reuse pair occurs. The LCAF is found using the activation tree:

Definition 5: The activation tree [7] of a running program is a tree with a node for every function call at runtime and edges pointing from callers to callees. □

The use site of a reuse pair (a_x, a'_x) is the node corresponding to the function invocation in which access $a_{x,}$ occurs. The reuse site is the node where access a'_x occurs. The Least Common Ancestor Frame (LCAF) of a reuse pair (a_x, a'_x) is the least common ancestor in the activation tree of the use site and the reuse site of (a_x, a'_x). The Least Common Ancestor Function is the function that corresponds to the least common ancestor frame.

Since loops are the most prominent places for data reuse, we look for the loops that carry the reuses. Similar to the least common ancestor function, the outermost loop is determined at which the use or the reuse occurs. The corresponding basic blocks are called the non-nested reuse basic blocks. These are defined as follows (see Figure 4).

Definition 6: *The* ***Nested Loop Forest*** *of a function is a graph, where each node represents a basic block in the function, and there are other edges from a loop header to each basic block directly controlled by that loop header. The* ***Outermost Executed Loop Header*** *(OELH) of a basic block BB with respect to a given reuse pair (a_x, a'_x) is the unique ancestor of that BB in the nested loop forest that has been executed between use a_x and reuse a'_x , but does not have ancestors itself that are executed between use and reuse. The* ***Non-nested Use Basic Block*** *(NNUBB) of (a_x, a'_x) is the OELH of the us basic block of (a_x, a'_x) .The* ***Non-nested Reuse Basic Block*** *(NNRBB) of (a_x, a'_x) is te OELH of the reuse basic block of (a_x, a'_x).* □

In practice, there are two major cases: the non-nested use and reuse basic blocks are the same, which means that the use and reuse basic blocks are contained within the a single outermost loop, or they are different, which means that the use and the use basic blocks are contained in different loops.

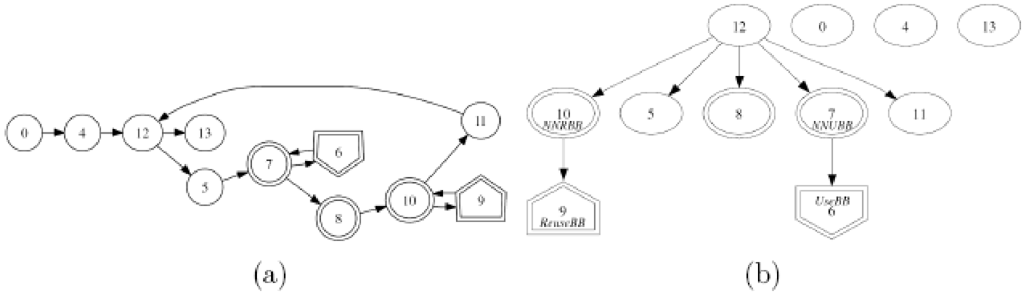


Figure 4: (a) A Control flow graph, and (b) corresponding loop forest. The basic blocks executed between use and reuse are indicated by double ellipses. For this particular reuse pair, the Use Basic Block is 6 and the Reuse Basic Block is 9.

When the use and the reuse basic blocks belong to a common loop, i.e., NNUBB=NNRBB, the reuse is contained in a single loop. A long reuse distance suggests that the loop traverses a large data structure in each iteration of the outer loop. The distance can be made smaller by ensuring that only a small part of the data structure is traversed in each iteration. The idea is to bring small chunks or tiles of data into the cache and making sure that all the operations on that chunk are executed before the next chunk is moved in. A number of transformations have been proposed, which tile large data structures to improve the locality: loop tiling [8], data shackling [9], time skewing [10], loop chunking [11]. A special case is loop permutation, where the outer and inner loops are swapped. In this way, long distances in the outer loop become small distances in the inner loop. These transformations are suggested by annotating the loops with the words TILE.

If both use and reuse are in different loops altogether, this indicates that the same data structure is accessed in one loop and again in the second loop. Locality can be improved by bringing these loops together so that the large data structure is accessed only a single time. This type of refactoring is suggested by the words FUSE.

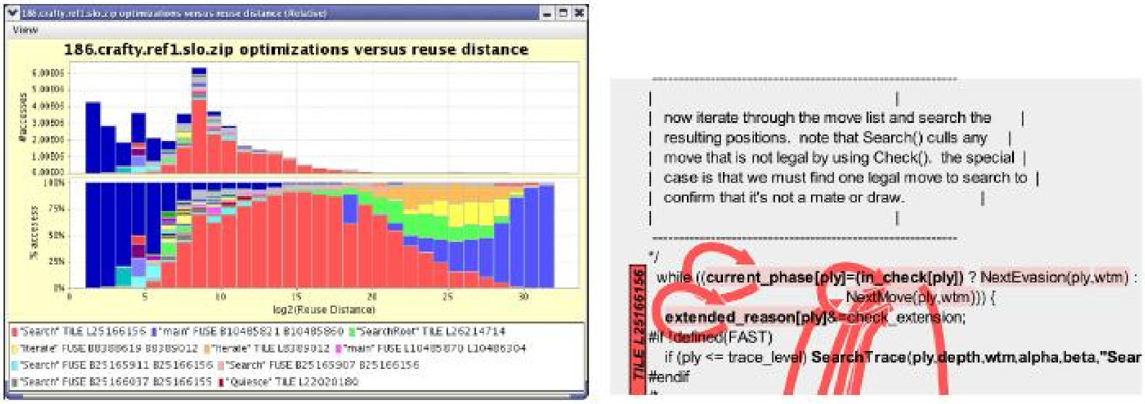
6. Results of Cache Hints Based Refactoring

Using RDVIS and SLO, a number of SPEC 2000 benchmarks were refactored for better cache behavior and data locality. The program 183.equake simulates earth quakes. The main optimization indicated was to tile an outer loop. Program 179.art simulates a neural network to recognize objects in an image, together with a confidence level of how sure it is the object is truly recognized. For this program, the tool shows that a middle loop needs to be tiled and a number of loop nests in that loop need to be fused. For most of the indicated refactorings, in both 183.equake and 179.art, naively applying them would have violated data dependences, resulting in incorrect output. Therefore, we applied a series of enabling transformations first to make the indicated refactorings legal. For both 183.equake and 179.art, some array data needed to be duplicated to eliminate false dependencies. Furthermore, a number of "enabling" loop transformations were required to make the indicated transformations legal. For 179.art, the tiling of the middle loop could not be performed even after trying to find enabling transformation, because of true data dependences. The output and visualization by the tools presented in this paper directed us towards finding a sequence of optimizations to improve data locality. After the transformations, the programs run more than two times faster on average on a number of different platforms, see Table 1.

	PENTIUM 4 (512KB)	ITANIUM (2MB)	ALPHA (8MB)	AVERAGE
Art	4.11	1.54	1.16	2.39
Equake	1.10	2.93	3.09	2.30

Table 1: Speedups on different platforms of Art and Equake after temporal locality optimizations performed based on suggestions made by RDVIS and SLO. The cache sizes of the largest cache level are indicated between parentheses for each platform.

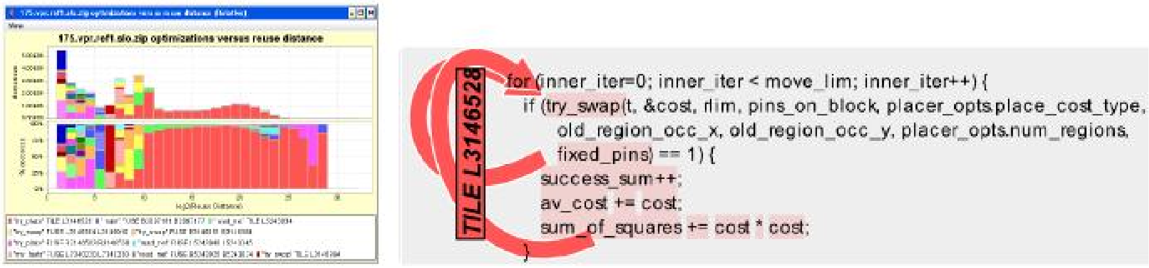
As an illustration of the power of the SLO tool in revealing the causes of poor temporal locality, we show two more examples from SPEC2000.


186.crafty.ref1.slo.zip optimizations versus reuse distance (Relative)
View
186.crafty.ref1.slo.zip optimizations versus reuse distance
#accesses
6.0000E6
5.0000E6
4.0000E6
3.0000E6
2.0000E6
1.0000E6
0.0000E0
100%
75%
50%
25%
0%
% accesses
0
5
10
15
20
25
30
log2(Reuse Distance)
"Search" TILE L25166156
"main" FUSE B10485821 B10485860
"SearchRoot" TILE L26214714
"Iterate" FUSE B8388619 B8389012
"Iterate" TILE L8389012
"main" FUSE L10485870 L10486304
"Search" FUSE B25165911 B25166156
"Search" FUSE B25165907 B25166156
"Search" FUSE B25166037 B25166155
"Quiesce" TILE L22020180
TILE L25166156

```
    ----------------------------------------------------------------
    |                                               |
    |  now iterate through the move list and search the      |
    |  resulting positions.  note that Search() culls any    |
    |  move that is not legal by using Check().  the special |
    |  case is that we must find one legal move to search to |
    |  confirm that it's not a mate or draw.               |
    |                                               |
    ----------------------------------------------------------------
*/
  while ((current_phase[ply]=(in_check[ply]) ? NextEvasion(ply,wtm) :
                                 NextMove(ply,wtm))) {
    extended_reason[ply]&=check_extension;
#if !defined(FAST)
    if (ply <= trace_level) SearchTrace(ply,depth,wtm,alpha,beta,"Sear
#endif
```

Figure 5: Program 186.crafty: SLO views. The main (red) optimization requires tiling the loop that iterates over all possible moves on the chess-board in a given board position.

Figure 5 depicts the major long-distance reuses for the chess program Crafty. This shows that the major refactoring required is tiling the loop that iterates over the list of possible moves of chess pieces for a given board position. Figure 6 shows the results for VPR, a place-and-route tool for FPGAs. From the figure it follows that most long-distance reuses occur between different invocations of `try_swap`, which optimizes placement by swapping two CLBs.


175.vpr.ref1.slo.zip optimizations versus reuse distance (Relative)
View
175.vpr.ref1.slo.zip optimizations versus reuse distance
#accesses
% accesses
0
5
10
15
20
25
30
log2(Reuse Distance)
TILE L3146528

```
for (inner_iter=0; inner_iter < move_lim; inner_iter++) {
  if (try_swap(t, &cost, rlim, pins_on_block, placer_opts.place_cost_type,
      old_region_occ_x, old_region_occ_y, placer_opts.num_regions,
      fixed_pins) == 1) {
    success_sum++;
    av_cost += cost;
    sum_of_squares += cost * cost;
}
```

Figure 6: Program 175.vpr: SLO views. VPR is a place-and-router for FPGA design. It shows the most important (red) refactoring in the source code for the placement phase in FPGA place-and-route.

7. Conclusions

Measuring the reuse distance is a key factor for avoiding capacity misses and improving the cache performance of regular programs. Using Presburger formulas to model the iteration space, the reuse distance can be calculated analytically and expressed by Ehrhart polynomials for simple loops, thereby giving a good insight into the relation between these distances and capacity misses. Carrying this a step further, a number of tools is presented, in which the reuse distance was calculated and mapped onto the intermediately executed code between use and reuse. Next, the most prominent loops are automatically selected and annotated with hints to improve the data locality by tiling or fusion. Using these techniques for a number of SPEC2000 benchmark programs results in a speed up of more than two. The tools developed are RDVIS, the reuse distance visualizer, and SLO, which Suggests Locality Optimizations. They are available at `http://www.elis.UGent.be/~kbeyls/rdvis` and `http://slo.sourceforge.net`, respectively.

Acknowledgments

Kristof Beyls was supported by research project GOA-12051002.

References

[1] K. Beyls, “Software Methods to Improve Data Locality and Cache Behavior”, *PhD thesis,* Ghent university, June 2004

[2] P. Clauss, “Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to analyze and Transform Scientific Programs,” *Proceedings of the International Conference on Supercomputing*, pp. 278-285, June 1996.

[3] S Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, “Analytical Computation of Erhart Polynomials: Enabling More Compiler Analyses and Optimizations”, *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 248-258, September 2004.

[4] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, “Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions”, *Algorithmica*, vol. 48, no. 1, 37-66, 2007

[5] K. Beyls and E.H. D’Hollander, “Generating Cache Hints for Improved Program Efficiency”, *Journal of Systems Architecture*, vol. 51, no. 4, pp. 223-250, 2005.

[6] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behavior”, *Proceedings of the 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45-57, Oct. 2002.

[7] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[8] J. Xue, *Loop Tiling for Parallelism*, Kluwer Academic Publishers. 2000

[9] I. Kodukula and K. Pingali, “Data-Centric Transformations for Locality Enhancement”. *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 319–364, 2001.

[10] D. Wonnacott, “Achieving Scalable Locality with Time Skewing”, *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, 2002.

[11] C. Bastoul and P. Feautrier, “Improving Data Locality by Chunking”, *Proceedings of the 12th International Conference on Compiler Construction,* LNCS 2622, pages 320–335, April 2003.