Intermediately Executed Code is the Key to Find Refactorings that Improve Temporal Data Locality

Kristof Beyls
Electronics and Information Systems (ELIS),
Ghent University,
Sint-Pietersnieuwstraat 41,
B-9000 Gent, Belgium
kristof.beyls@elis.UGent.be

Erik H. D'Hollander
Electronics and Information Systems (ELIS)
Ghent University,
Sint-Pietersnieuwstraat 41,
B-9000 Gent, Belgium
erik.dhollander@elis.UGent.be

ABSTRACT

The growing speed gap between memory and processor makes an efficient use of the cache ever more important to reach high performance. One of the most important ways to improve cache behavior is to increase the data locality. While many cache analysis tools have been developed, most of them only indicate the locations in the code where cache misses occur. Often, optimizing the program, even after pinpointing the cache bottlenecks in the source code, remains hard with these tools.

In this paper, we present two related tools that not only pinpoint the locations of cache misses, but also suggest source code refactorings which improve temporal locality and thereby eliminate the majority of the cache misses. In both tools, the key to find the appropriate refactorings is an analysis of the code executed between a data use and the next use of the same data, which we call the Intermediately Executed Code (IEC). The first tool, the Reuse Distance VISualizer (RDVIS), performs a clustering on the IECs, which reduces the amount of work to find required refactorings. The second tool, SLO (short for "Suggestions for Locality Optimizations"), suggests a number of refactorings by analyzing the call graph and loop structure of the IEC. Using these tools, we have pinpointed the most important optimizations for a number of SPEC2000 programs, resulting in an average speedup of 2.3 on a number of different platforms.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers,Debuggers,Optimization*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Performance, Measurement, Languages

Keywords

Temporal Data Locality, Program Analysis, Refactoring, Program Optimizations, Performance Debugger, Loop Transformations

1. INTRODUCTION

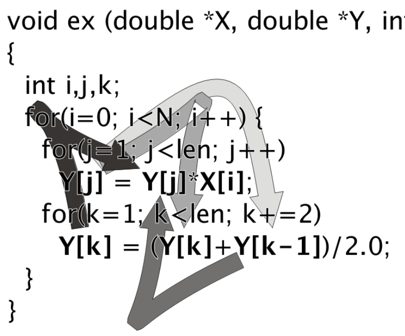
The widening speed gap between processor and main memory makes low cache miss rates ever more important. The major classes of cache misses are conflict and capacity misses. While conflict misses are caused by conflicts in the internal cache structure, capacity misses are caused by poor temporal or spatial locality. In this paper, we propose two tools that help to identify the underlying reason of poor temporal data locality in the source code.

1.1 Related Work

In recent years, compiler methods have been devised to automatically increase spatial data locality, by transforming the data layout of arrays and structures, so that data accessed close together in time also lays close together in the address space [9, 11, 17, 19, 22, 33]. On the other hand, temporal locality can only be improved by reordering the memory accesses so that the same addresses are accessed closer together. Advanced compiler methods to do this all target specific code patterns such as affine array expressions in regular loop nests [11, 18, 22], or specific sparse matrix computations [14, 15, 24, 27]. For more general program constructs, fully-automatic optimization seems to be very hard, mainly due to the difficulty of the required dependence analysis. Therefore, cache and data locality analysis tools and visualizers are needed to help programmers to refactor their programs for improved temporal locality.

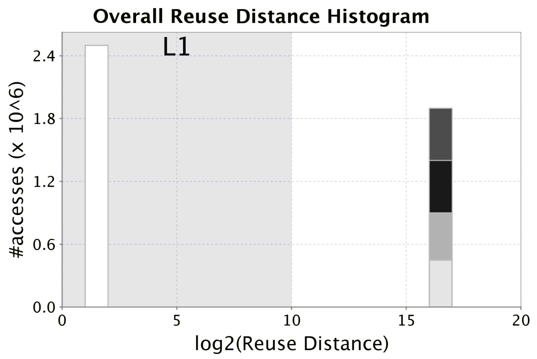
```
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j]=Y[j]*X[i];          // 39% of cache misses
    for(k=1; k<len; k+=2)
      Y[k]=(Y[k]+Y[k-1])/2.0;// 61% of cache misses
  }
}
```

Figure 1: First motivating example, view on cache misses given by traditional tools. N=10, len=100001.

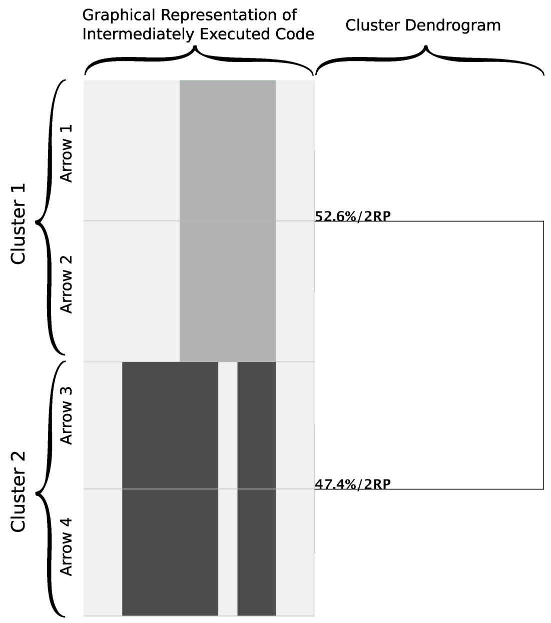


```
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j] = Y[j]*X[i];
    for(k=1; k<len; k+=2)
      Y[k] = (Y[k]+Y[k-1])/2.0;
  }
}
```

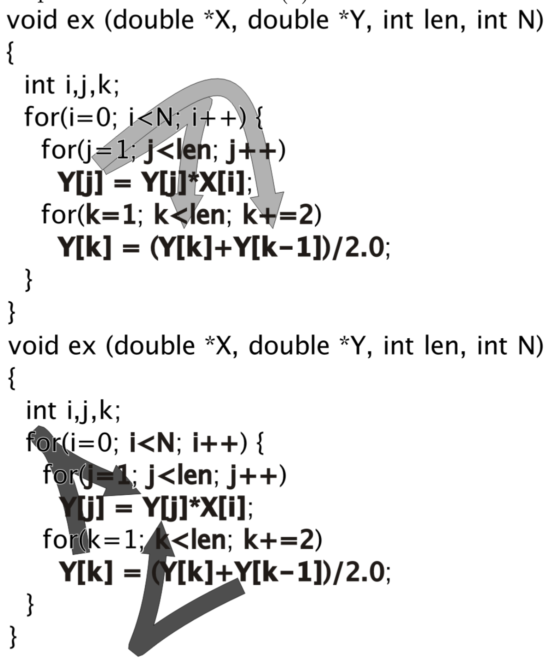
(a) view of reference pairs with long-distance reuse in RDVIS.



(b) histogram of long-distance reuses. Gray scales correspond to the arrows in (a).



(c) graphical view of intermediately executed code in RDVIS, and associated cluster analysis.



```
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j] = Y[j]*X[i];
    for(k=1; k<len; k+=2)
      Y[k] = (Y[k]+Y[k-1])/2.0;
  }
}
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j] = Y[j]*X[i];
    for(k=1; k<len; k+=2)
      Y[k] = (Y[k]+Y[k-1])/2.0;
  }
}
```

(d) view of Intermediately Executed Code of resp. the light and the dark gray cluster in (c).

Figure 2: First motivating example, views produced by RDVIS. The colors were manually changed to gray scale, to make the results readable in this black-and-white copy of the paper.

Most existing cache and data locality analyzers measure the locality or the cache misses and indicate at which locations in the source code, or for which data structures, most cache misses occur [2, 4, 5, 8, 12, 13, 20, 21, 23, 28, 29, 32]. While this information is helpful in identifying the main bottlenecks in the program, it can still be difficult to deduce a suitable program transformation from it. In this regard, a few of these tools provide additional support for finding the underlying cause of conflict misses (e.g. CVT[28], CacheVIZ[32], YACO[25]) or the underlying cause of poor spatial locality (e.g. SIP[4]).

In contrast, we present a method to help identify the underlying causes of poor *temporal* data locality. Basically, poor temporal locality results when a large amount of other data is accessed between two consecutive uses of the same data. Improving the locality requires diminishing the volume of data accessed between use and reuse. The source code executed between use and reuse is responsible for accessing the large data volume, resulting in a *long reuse distance*. That source code is called the *Intermediately Executed Code (IEC)* of that reuse. Consequently, to improve the temporal data locality, a refactoring of the IEC is required.

In this paper, we present two tools that analyze the IEC in different ways to pinpoint the required refactorings: *RDVIS (Reuse Distance VISualizer)*, which has been discussed earlier in [7], and *SLO (Suggestions for Locality Optimizations)*. RDVIS represents the IEC as a set of basic blocks executed between long-distance reuses. In a typical program, there are a huge number of data reuses, and consequently a huge number of corresponding IECs. RDVIS applies a cluster analysis to the IECs so that the main patterns of poor locality-generating source code are revealed. Based on a visual representation of the resulting clusters and highlighting of the corresponding source code, the programmer can deduce the necessary program optimizations. In SLO, the loop structure and the call graph of the IEC is also taken into account, allowing it to go one step further than RDVIS. SLO pinpoints the exact source code refactorings that are needed to improve locality. Examples of such refactorings are loop tiling, and computation fusion, which are demonstrated in the motivating examples in section 2. In section 3, reuse distances and associated terms are defined. Section 4 describes how RDVIS analyzes the IEC. Section 5 presents the analyses performed by SLO on the IEC to find the appropriate source code refactorings. In section 6, we provide a few case studies where these tools have been used to identify the required refactorings for a number of real-world programs from the SPEC2000 benchmarks. For two of them, we applied the necessary transformations, leading to an average cross-platform speedup of about 2.3. Concluding remarks are given in section 7.

2. MOTIVATING EXAMPLES

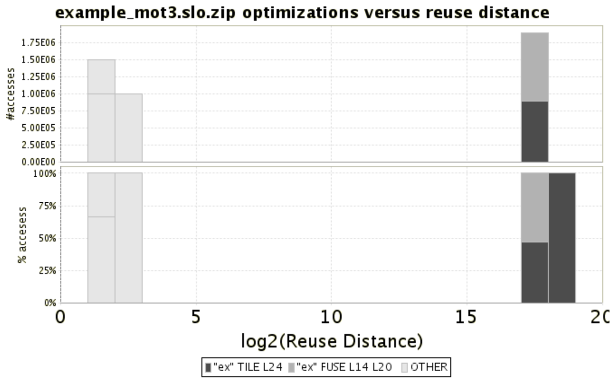
We start by showing two small code examples where the indication of cache misses with traditional tools does not clearly reveal how to optimize the programs. Furthermore, we show how RDVIS and SLO visualize the intermediately executed code of long-distance reuses, and how that makes it easier to find source code refactorings that improve temporal locality.

2.1 Example 1: Intra-Procedural Loop Reuses

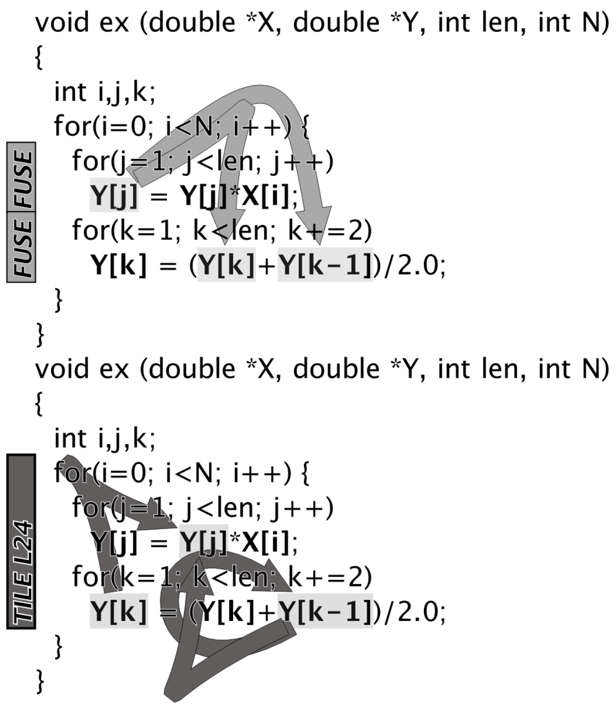
The code in figure 1 shows a small piece of code, where a traditional tool would show that the first statement is responsible for about 39% of all cache misses and the second statement produces 61% of them. While this information indicates where cache misses occur, it is not directly clear how the locality of the program can be improved to diminish the number of cache misses.

The views produced by our tools are shown in figure 2 for RDVIS, and in figure 3 for SLO. For each pair of references that generate many long-distance reuses, an arrow is drawn, starting at the reference that accesses the data first, and pointing to the reference that reuses that data after a long time. Figure 2(a) shows the four pairs of references that generate the majority of long-distance reuses: (`Y[k]`, `Y[j]`), (`Y[k-1]`, `Y[j]`), (`Y[j]`, `Y[k]`) and (`Y[j]`, `Y[k-1]`). Figure 2(b) shows that each of those 4 pairs generate about the same amount of long-distance reuses at distance 2^{17}, meaning that about 2^{17} other elements are accessed between those reuses. (In this example, `N`=10 and `len`=100001). When the cache can contain 2^{10} elements (as indicated by the background), all the reuses at a larger distance lead to cache misses. So, the reuses at distance 2^{17} must be made smaller than 2^{10}, in other words, largely diminishing the amount of data accessed between use and reuse.

To optimize each of the four arrows in figure 2(a), the first step is to pinpoint which code is responsible for generating the accesses between use and reuse. The second step is to refactor the code so that fewer data elements are accessed between use and reuse. RDVIS records the basic blocks executed between each use and reuse, and allows to visually indicate the corresponding source code for each arrow. Besides an evaluation examining each arrow separately, RDVIS also contains a cluster analysis. The arrows with similar IEC are put in the same cluster. As an example, figure 2(c) shows how RDVIS graphically represents the result of the cluster analysis. On the left hand side, the code executed



(a) 5 different optimizations indicated by gray scale, with respect to the reuse distance of the reuses they optimize, as shown by SLO

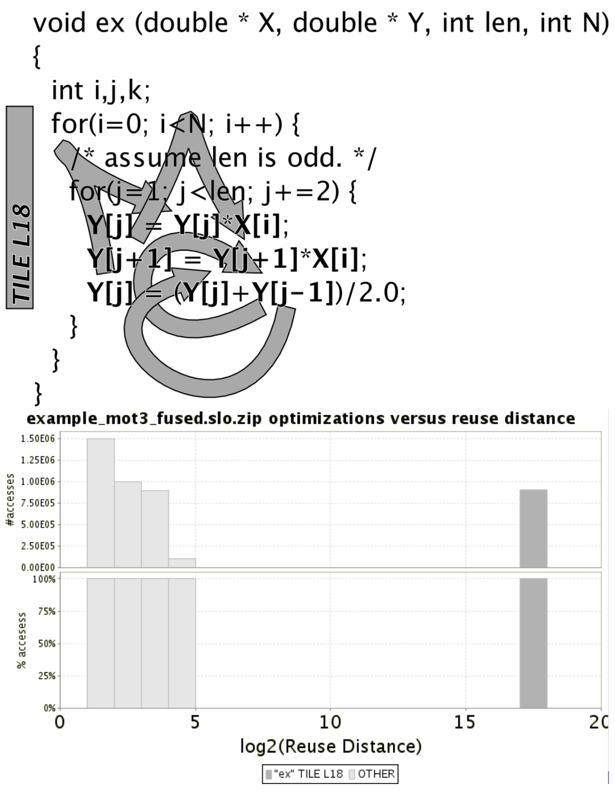


```
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j] = Y[j]*X[i];
    for(k=1; k<len; k+=2)
      Y[k] = (Y[k]+Y[k-1])/2.0;
  }
}
void ex (double *X, double *Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
    for(j=1; j<len; j++)
      Y[j] = Y[j]*X[i];
    for(k=1; k<len; k+=2)
      Y[k] = (Y[k]+Y[k-1])/2.0;
  }
}
```

(b) Indication of the two optimizations for the reuses at distance 2^{17}, as indicated by SLO. The light gray optimization indicates fusion of the two inner loops. The dark gray optimization requires tiling the outer loop.

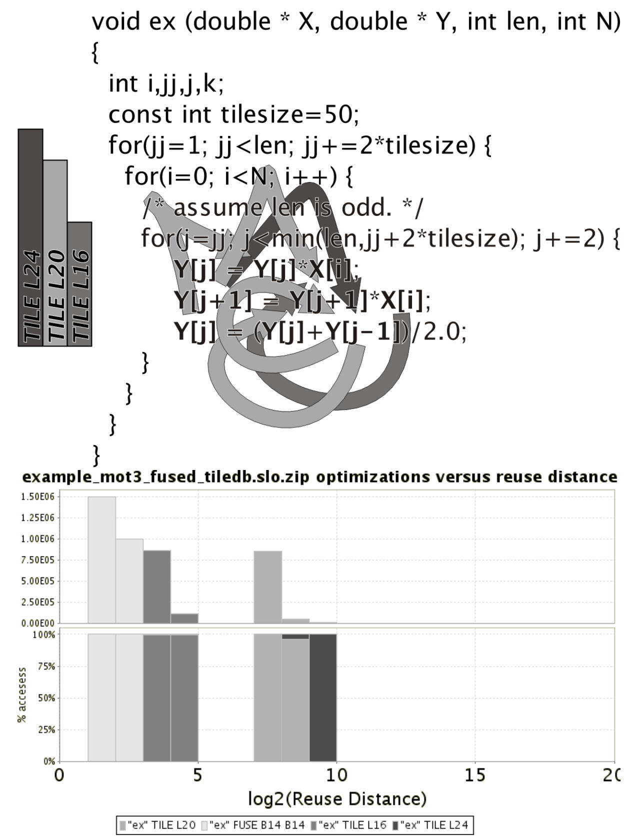
Figure 3: First motivating example, SLO view. The colors were manually altered to gray scale, to make the results readable in this black-and-white copy of the paper.

between use and reuse is graphically represented. There are four horizontal bands, respectively representing the IEC of the four arrows in figure 2(a). In each band, the basic blocks in the program are represented, left to right. If a basic block is executed between use and reuse, it is colored in a shade of gray, otherwise it is nearly white. Originally, RDVIS produces a colored view with higher contrast. Here, the colors were converted to enhance readability in black-and-white. Figure 2(c) shows that the code executed between use and reuse of arrows 1 and 2 are identical. Also the code exe-



```
void ex (double * X, double * Y, int len, int N)
{
  int i,j,k;
  for(i=0; i<N; i++) {
   /* assume len is odd. */
   for(j=1; j<len; j+=2) {
     Y[j] = Y[j]*X[i];
     Y[j+1] = Y[j+1]*X[i];
     Y[j] = (Y[j]+Y[j-1])/2.0;
   }
  }
}
```

Figure 4: Code and left-over long reuse distance after loop fusion.



```
void ex (double * X, double * Y, int len, int N)
{
  int i,jj,j,k;
  const int tilesize=50;
  for(jj=1; jj<len; jj+=2*tilesize) {
    for(i=0; i<N; i++) {
      /* assume len is odd. */
      for(j=jj; j<min(len,jj+2*tilesize); j+=2) {
        Y[j] = Y[j]*X[i];
        Y[j+1] = Y[j+1]*X[i];
        Y[j] = (Y[j]+Y[j-1])/2.0;
      }
    }
  }
}
```

Figure 5: Code and left-over long reuse distance after loop tiling.

cuted between use and reuse of arrow 3 and 4 are identical. On the right hand side, the cluster dendrogram graphically indicates how "similar" the IEC is for each arrow. In this example, the user has manually selected two subclusters. It shows that 52.6% of the long distance reuses are generated by the light gray cluster, while 47.4% are generated by the dark gray cluster. Furthermore, in figure 2(d), the IEC for the two clusters has been highlighted in the source code by RDVIS. The code that is executed between use and reuse is highlighted in bold. This shows that for the light gray cluster, the uses occur in the `j`-loop, while the reuses occur in the `k`-loop. Both the use and the reuse occur in the same iteration of the `i`-loop, since the loop control code: `i<N; i++` is not highlighted. These two arrows can be optimized by *loop fusion*, as is discussed in detail below. In the dark gray cluster, it shows that the control of loop i: `i<N; i++` is executed between use and reuse. Hence the use and reuse occur in different iterations of the outer `i`-loop. The experienced RDVIS-user recognizes from this pattern that *loop tiling* needs to be applied, as discussed in more detail below.

In contrast to RDVIS, where the programmer needs to examine the Intermediately Executed Code to pinpoint optimizations, SLO analyzes the IEC itself, and interactively indicates the optimizations that are needed. For example, in figure 3(b), the required loop fusion and loop tiling are indicated by a bar on the left hand side. Furthermore, the histogram produced by SLO indicates which reuses can be optimized by which optimization in different colors, e.g. see figure 3(a). The upper histogram shows the absolute number of reuses at a given distance. The bottom histogram shows the fraction of reuses at a given distance that can be optimized by each transformation.

Below, we explain how loop fusion and loop tiling can be used to improve the locality and performance. These two transformations are the most important optimizations for improving temporal locality in loops.

2.1.1 Optimizing Pattern 1: Loop Fusion

From both the views produced by RDVIS (fig. 2(d) at the top) and SLO (fig. 3(b) at the top), it shows that about half of the long-distance reuses occur because element `Y[j]` is used in the first loop, and it is reused by references `Y[k]` and `Y[k-1]` in the second loop. The distance is long because between the reuses, all other elements of array `Y` are accessed by the same loops. For this pattern, the reuse distance can be reduced by *loop fusion*: instead of running over array `Y` twice, the computations from both loops are performed in one run over the array. In order to fuse the loops, the first loop is unrolled twice, after which they are fused, under the assumption that variable `len` is odd, resulting in the code in figure 4. The histogram in the figure shows that the long-distance reuses targeted have all been shortened to distances smaller than 2^5. This results in a speedup of about 1.9 on a Pentium4 system, due to fewer cache misses, see table 1.

2.1.2 Optimizing Pattern 2: Loop Tiling

After fusing the inner loops, the code can be analyzed again for the causes of the remaining long reuse distance patterns. Figure 4 shows how SLO indicates that all left-

version	exec. time	speedup
orig	0.183s	
fused	0.098s	1.87
fused+tiled	0.032s	5.72

Table 1: Running times and speedups of the code before and after optimizations, on a 2.66Ghz Pentium4, for N=10, len=1000001.

```
1  double inproduct(double *X, double *Y, int len) {
     int i; double result=0.0;
     for(i=0; i<len; i++)
       result += X[i]*Y[i]; // 50% of cache misses
5    return result;
   }

   double sum(double *X, int len) {
     int i; double result=0.0;
10   for(i=0; i<len; i++)
       result += X[i];      // 50% of cache misses
     return result;
   }

15 double prodsum(double *X, double *Y, int len) {
     double inp = inproduct (X,Y,len);
     double sumX = sum (X,len);
     double sumY = sum (Y,len);
     return inp+sumX+sumY;
20 }
```

Figure 6: View on cache misses as provided by most traditional tools for the second example.

over long reuse distances occur because the use is in one iteration of the i-loop, and the reuse is in a later iteration. Consequently, the tool indicates that the i-loop should be tiled, by displaying a bar to the left of the loop source code.

Loop tiling is applied when the long-distance reuses occur between different iterations of a single outer loop. When this occurs, it means that in a single iteration of that loop, more data is accessed than can fit in the cache. The principle idea behind loop tiling is to process less data in one iteration of the loop, so that data can be retained in the cache between several iterations of the loop. Figure 5 shows the code after tiling. Now, the inner j-loop executes at most 50 iterations (see variable tilesize), and hence the amount of data accessed in the inner loop is limited. As a result, the reuses between different iterations of the i-loop are shortened from a distance of 2^{17} to a distance between 2^7 and 2^9, see the histograms in Figures 4 and 5. Note that some reuses have increased in size: 1 in 50 reuses between iterations of the j-loop in figure 4 have increased from 2^4–2^5 to 2^9–2^{10} (see dark bars in figure 5). This is because 1 in 50 reuses in the original j-loop are now between iterations of the outer jj-loop. The end result is the removal of all long-distance reuses. As a result, the overall measured program speedup is 5.7, see table 1.

2.2 Example 2: Inter-Procedural Reuses

The second example is shown in figure 6. The code in function prodsum first calculates the inproduct of two arrays by calling inproduct, after which the sum of all elements in both arrays is computed by calling function sum. Most existing tools would show, in one way or another, that half of the misses occur on line 4, and the other half are caused by the code on line 11.

In contrast, RDVIS shows two reference pairs, indicated by arrows, that lead to long distance reuses, see figure 7. By examining the highlighted code carefully, the programmer can find that uses occur in the call to inproduct, while reuses occur in one of the two calls to sum. Here, the programmer must perform an interprocedural analysis of the IEC. SLO, on the other hand, performs the interprocedural analysis for the programmer, and visualizes the result as shown in figure 8. It clearly identifies that for half of the long-distances reuses, inproduct must be fused with the first call to sum, and for the other half inproduct must be fused with the second call to sum.

3. BASIC DEFINITIONS

In this section, we review the basic terms and definitions that are used 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 that leads to a memory instruction at compile-time, which in turn generates memory accesses at run-time. The reference that generates memory access a_x is denoted by ref(a_x). The address accessed be a memory access is denoted by addr(a_x), i.e. addr(a_x) $= x$.

Definition 2. A **memory access trace** $\mathcal{T}$ 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 3. A **reuse pair** $\langle a_x, a'_x\rangle$ 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** (r_1, r_2) is a pair of memory references. The reuse pairs associated with a reference pair (r_1, r_2) is the set of reuse pairs for which the use is generated by r_1 and the reuse is generated by r_2, and is denoted by reuses(r_1, r_2).

Definition 4. The **Intermediately Executed Code (IEC)** of a reuse pair $\langle a_x, a'_x\rangle$ is the code executed between $T[a_x]$ and $T[a'_x]$.

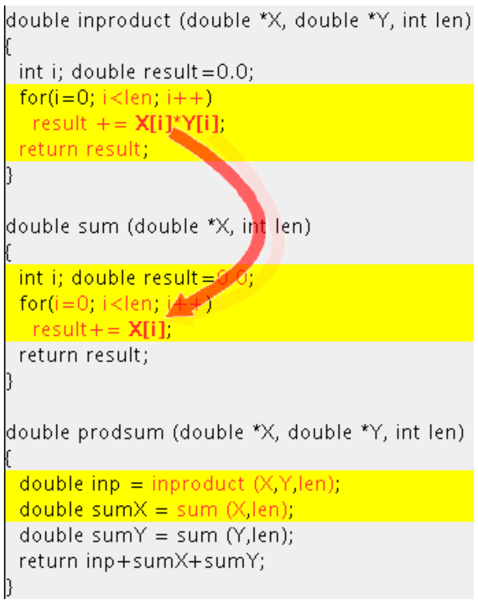
Definition 5. The **reuse distance** of a reuse pair from a trace, is the number of unique memory addresses in that trace between use and reuse.

Cache misses are identified by the reuses that have a distance larger than the cache size [6].

4. RDVIS: IEC ANALYSIS BY BASIC BLOCK VECTOR CLUSTERING

In RDVIS, the Intermediately Executed Code is represented by a basic block vector:

Definition 6. The **basic block vector of a reuse pair** $\langle a_x, a'_x\rangle$**, denoted by BBV**$(\langle a_x, a'_x\rangle)$ is a vector $\in \{0,1\}^n$, where n is the number of basic blocks in the program. When a basic block is executed between use and reuse, the corresponding vector element is 1, otherwise it is 0.

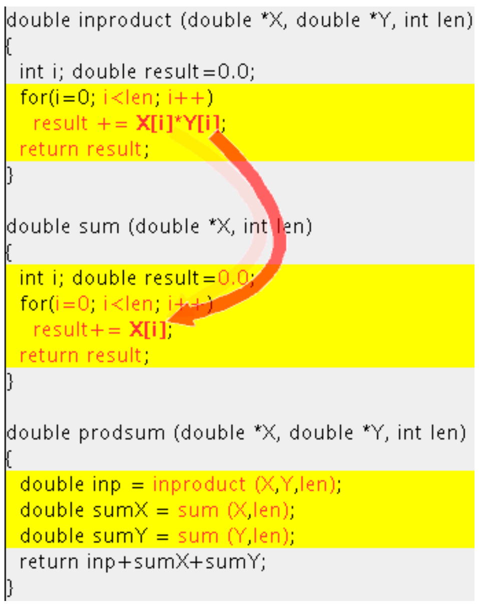


```
double inproduct (double *X, double *Y, int len)
{
  int i; double result=0.0;
  for(i=0; i<len; i++)
    result += X[i]*Y[i];
  return result;
}

double sum (double *X, int len)
{
  int i; double result=0.0;
  for(i=0; i<len; i++)
    result+= X[i];
  return result;
}

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

(a) IEC for first reference pair.



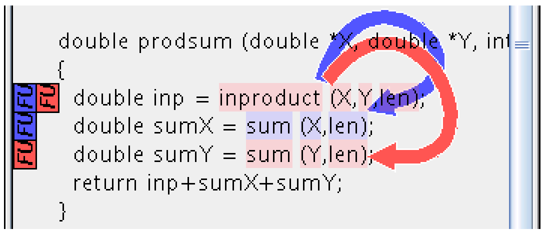
```
double inproduct (double *X, double *Y, int len)
{
  int i; double result=0.0;
  for(i=0; i<len; i++)
    result += X[i]*Y[i];
  return result;
}

double sum (double *X, int len)
{
  int i; double result=0.0;
  for(i=0; i<len; i++)
    result+= X[i];
  return result;
}

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

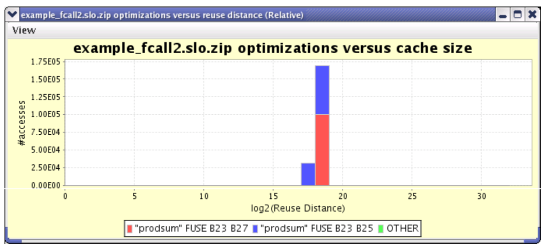
(b) IEC for second reference pair.

Figure 7: Indication of intermediately executed code by RDVIS.



```
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;
}
```

(a) Two required fusions of functions indicated by arrows.



(b) The reuse distance histogram for the reuses optimized by the two arrows in (a), for `len`=1000000.

Figure 8: Indication of locality optimizations by SLO.

The **basic block vector of a reference pair** (r_1, r_2)**, denoted by BBV**$((r_1, r_2))$ is a vector $\in [0, 1]^n$. The value of a vector element is the fraction of reuse pairs in reuses(r_1, r_2) for which the basic block is executed between use and reuse. More formally:

$$BBV((r_1, r_2)) = \frac{\sum_{\langle a_x, a'_x \rangle \in \text{reuses}(r_1, r_2)} \text{BBV}(\langle a_x, a'_x \rangle)}{\#\text{reuses}(r_1, r_2)}$$

In RDVIS, reference pairs are visually represented by arrows drawn on top of the source code, e.g. figure 2. The tool allows to highlight the code executed between use and reuse for each individual arrow. Additionally, RDVIS clusters arrows according to the similarity of their IEC.

The similarity (or rather dissimilarity) of the code executed between two reference pairs is computed as the Manhattan distance of the corresponding basic block vectors in the vector space $[0, 1]^n$. When exactly the same code is executed between the reuses, the distance is 0; when the code is completely dissimilar, the distance is n. Based on the Manhattan distance, an agglomerative clustering is performed, which proceeds as follows. First, each reference pair forms a separate cluster. Then, iteratively, the two closest clusters are merged into a single cluster. The basic block vector corresponding with the new cluster is the average of the two basic block vectors that represent the merged clusters. The clustering stops when all reference pairs are combined into one large cluster. The distances between different subclusters are shown graphically in the dendrogram, and the user selects "interesting-looking" or "tight" subclusters. E.g. in figure 2(c), the user selected two very tight subclusters: the light gray and the dark gray subcluster. Since similar code is executed between use and reuse in a tight subcluster, it is likely that the long-distance reference pairs can be optimized by the same refactoring, e.g. see figure 2(d).

5. SLO: IEC ANALYSIS BY INTERPROCEDURAL CONTROL FLOW INSPECTION

SLO aims to improve on RDVIS by analyzing the IEC further and automatically pinpoint the refactorings that are necessary to improve temporal locality, even in an interprocedural context. To make this possible, SLO tracks the loop headers (i.e. the basic blocks that control whether a loop body is executed [1]) and the functions that are executed between use and reuse, using the following framework.

5.1 Step 1: Determining the Least Common Ancestor Function

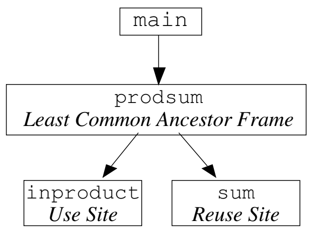


Figure 9: The Least Common Ancestor Frame (LCAF) of a reuse, indicated in the activation tree. The activation tree represents a given time during the execution of the code in figure 6, assuming that the use occurs inside function `inproduct`, and the reuse occurs inside `sum`.

SLO proceeds by first determining the function in which the refactoring must be applied. In a second step, the exact refactoring on which part of that function's code is computed. The refactoring must be applied in the "smallest" function in which both the use and the reuse can be seen. This is formalized by the following definitions, and illustrated in figure 9.

Definition 7. The **activation tree** [1] of a running program is a tree with a node for every function call at run-time and edges pointing from callers to callees.

The **use site** of a reuse pair $\langle a_x, a'_x \rangle$ 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 $\langle a_x, a'_x \rangle$ is the least common ancestor in the activation tree of the use site and the reuse site of $\langle a_x, a'_x \rangle$. The **Least Common Ancestor Function** is the function that corresponds to the least common ancestor frame.

The LCAF is the function where some refactoring is needed to bring use and reuse closer together. Once the LCAF has been determined, the loop structure of the LCAF is examined, and the basic blocks in the LCAF executed between use and reuse.

Definition 8. The basic block in the LCAF, in which the use occurred (directly or indirectly through a function call), is called the **Use Basic Block (UseBB) of $\langle \mathbf{a_x}, \mathbf{a'_x} \rangle$**; the basic block that contains the reuse is called the **Reuse Basic Block (ReuseBB) of $\langle \mathbf{a_x}, \mathbf{a'_x} \rangle$**.

5.2 Step 2: Analyzing the Control Flow Structure in the Least Common Ancestor Function

The key to the analysis is finding the loops that "carry" the reuses. These loops are found by determining the Non-nested Use and Non-nested Reuse Basic Blocks, as defined below (illustrated in figure 10):

Definition 9. The **Nested Loop Forest** of a function is a graph, where each node represents a basic block in the function, and there are 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** $\langle a_x, a'_x \rangle$ is the unique ancestor of 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 $\langle \mathbf{a_x}, \mathbf{a'_x} \rangle$** is the OELH of the use basic block of $\langle a_x, a'_x \rangle$. The **Non-nested Reuse Basic Block (NNRBB) of $\langle \mathbf{a_x}, \mathbf{a'_x} \rangle$** is the OELH of the reuse basic block of $\langle a_x, a'_x \rangle$.

5.3 Step 3: Determining the Required Refactoring

Refactorings are determined by analyzing the NNUBB and NNRBB. We subdivide in 3 different patterns:

Pattern 1: Reuse occurs between iterations of a single loop. This occurs when NNUBB = NNRBB, and they are loop headers. Consequently, a single loop carries the reuses. This pattern arises when the loop traverses a "data structure"[1] in every iteration of the loop. The distance of reuses across iterations can be made smaller by ensuring that only a small part of the data structure is traversed in any given iteration. As such, reuses of data elements between consecutive iterations are separated by only a small amount of data, instead of the complete data structure.

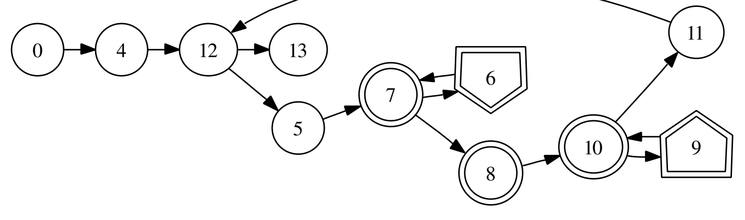
A number of transformations have been proposed to increase temporal locality in this way, e.g. loop tiling [26, 30], data shackling [18], time skewing [31], loop chunking [3], data tiling [16] and sparse tiling [27]. We call these transformations **tiling-like** optimizations. An extreme case of such a tiling-like optimization is loop permutation [22], where inner and outer loops are swapped, so that the long-distance accesses in different iterations of the outer loop become short-distance accesses between iterations of the inner loop.

Examples of occurrences of this pattern are indicated by bars with the word "TILE L..." in figures 3, 4 and 5.

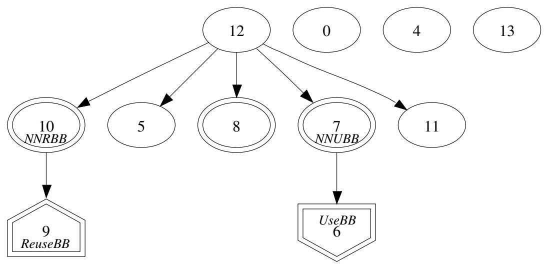
Pattern 2: Use is in one loop nest, the reuse in another. When NNUBB and NNRBB are different loop headers, reuses occur between different loops. The code traverses a data structure in the loop indicated by the NNUBB. The data structure is retraversed in the NNRBB-loop. The reuses can be brought closer together by only doing a single traversal, performing computations from both loops at the same time. This kind of optimization is known as loop fusion. We call the required transformation a **fusion-like** optimization. Examples of this pattern are indicated by bars with the word "FUSE L..." in figure 3.

Pattern 3: NNUBB and NNRBB are not both loop headers. When one of NNUBB or NNRBB are not loop headers, it means that either the use or the reuse is not inside a loop in the LCAF. It indicates that data is accessed in one basic block (possibly indirectly through a function call), and the other access may or may not be in a loop. So, the reused data structure is traversed twice by two separate code pieces. In this case, bringing use and reuse closer together requires that the computations done in the NNUBB and in the NNRBB are "fused" so that the data structure is

[1]the data structure could be as small a single scalar variable or as large as all the data in the program



(a) The Control Flow Graph.



(b) The Nested Loop Forest.

Figure 10: The Control Flow Graph and Nested Loop Forest of function ex in figure 1. 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.

traversed only once for performing both computations. As such, improving the temporal locality for this code pattern also requires a **fusion-like** optimization. An example of this pattern is indicated by bars with the word "FUSE B..." in figure 8.

Taking into account the three possible patterns of code leading to poor temporal locality, it is clear that the required transformations can be categorized in two classes:

tiling-like optimizations are required when reuses occur between iterations of a single loop. I.e. the loop mostly or completely traverses the same data structure in each of its iterations.

fusion-like optimizations are needed when two separate pieces of code, be it loops, function calls or plain non-loop code, access the same data.

6. IMPLEMENTATION AND CASE STUDIES

6.1 Implementation

We extended the GCC compiler to instrument all memory accesses, basic block transitions and function entry and exits. Additionally, the exact source code locations of the tokens constituting a memory reference and a basic block are recorded to file during compilation. At run-time, the instrumented program tracks all reuses and the intermediately executed code. At the end of execution, the instrumented program writes the recorded reuse information to disk.

This information is then read in by either RDVIS or SLO to visualize the major long-distance reuse patterns and associated intermediately executed code. RDVIS and the adapted GCC compiler are publicly available at `http://www.elis.ugent.be/~kbeyls/rdvis`. We plan to make SLO publicly available in the near future at `http://www.elis.ugent.be/~kbeyls/slo`.

6.2 Case studies

Using RDVIS and SLO, we examined and optimized a few programs from the SPEC2000 benchmark.

183.equake simulates earth quakes. The main optimization indicated was to tile an outer loop. We applied a special tiling akin the one presented in [27]. 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.

	Speedup			
	Pentium4 (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 2: 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.

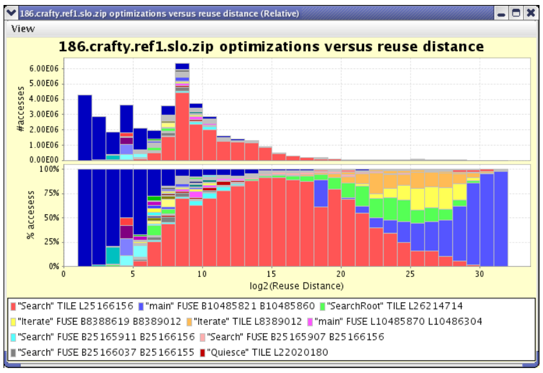
For most of the indicated refactorings, in both 183.equake and 179.art, naively applying them would have violated data dependences, resulting in incorrect output. Therefore, we applied a series of enabling transformations first to make the indicated refactorings legal. For both 183.equake and 179.art, 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.

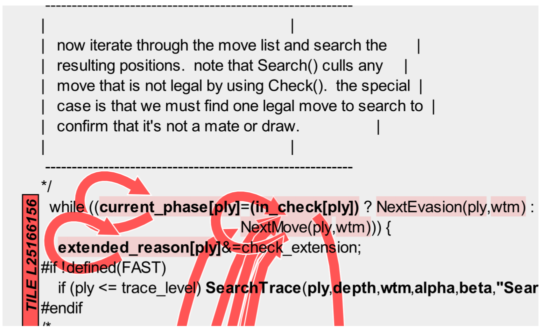
As such, the output and visualization by the tools presented in this paper directed us towards finding a sequence of optimizations to improve data locality [10]. After the transformations, the programs run more than 2 times faster on average on a number of different platforms, see table 2.

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. Figure 11 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 12 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 CLB's.

7. CONCLUSION

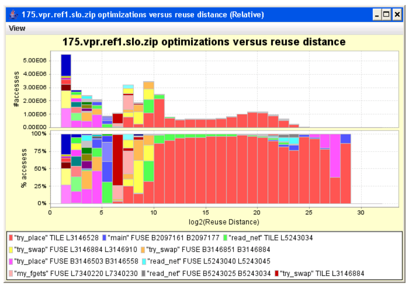
We have presented two tools that, to the furthest of our knowledge, are the first tools that identify refactorings which improve temporal data locality for arbitrary programs. Us-

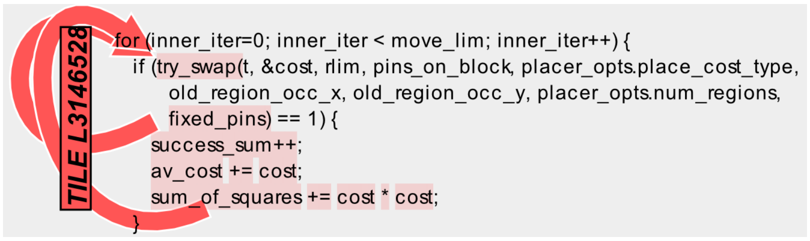




```
 ----------------------------------------------------------
 |                                                     |
 |  now iterate through the move list and search the     |
 |  resulting positions.  note that Search() culls any   |
 |  move that is not legal by using Check().  the special  |
 |  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 11: 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.





```
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 12: 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.

ing an extended GCC compiler, programs are instrumented and profiled. After profiling, an interactive visualizer allows to browse the most important refactorings revealed by the analyses. For both tools, the key observation towards finding suitable optimizations is that the code executed between a use and a reuse is responsible for generating a long-distance reuse. Eliminating the long-distance reuse requires accessing less data between the reuses, by refactoring the intermediately executed code (IEC). RDVIS helps by clustering the patterns of intermediately executed code, and directing programmers into examining the most important clusters of IEC. SLO additionally analyzes the interprocedural and loop structure of the IEC to automatically pinpoint the required refactoring, in the least common ancestor function (LCAF).

Our analysis reveals that all temporal data locality optimizations can be categorized as either tiling-like or fusion-like. A tiling-like optimization is required when data is reaccessed in subsequent iterations of the same loop. A fusion-like optimization is needed when a data structure is traversed twice by two different parts of the source code. Using the tools, we quickly pinpointed the underlying cause of poor temporal locality in a number of complex programs from the SPEC2000 benchmark. After taking into account data dependences, 2 SPEC2000 programs were optimized, resulting in an average speedup of 2.3 on a number of different platforms.

8. ACKNOWLEDGMENTS

Kristof Beyls was supported by research project GOA-12051002.

9. REFERENCES

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

[2] M. Atkins and R. Subramaniam. Pc software performance tuning. *IEEE Computer*, 29:47–54, 1996.

[3] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Compiler Construction, LNCS 2622*, pages 320–335, 2003.

[4] E. Berg and E. Hagersten. SIP: Performance tuning through source code interdependence. In *Euro-Par, LNCS 2400*, pages 177–186, 2002.

[5] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, pages 169–180, 2005.

[6] K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. of Systems Architecture*, 51(4):223–250, 2005.

[7] K. Beyls, E. H. D'Hollander, and F. Vandeputte. RDVIS: A tool that visualizes the causes of low locality and hints program optimizations. In *ICCS*, volume 3515 of *LNCS*, pages 166–173, 2005.

[8] R. Bosch and C. S. et al. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34:68–73, 2000.

[9] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI, SIGPLAN Notices*, pages 13–24, 1999.

[10] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for

compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05), Boston, Massachusetts.*, June 2005.

[11] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[12] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *PACT*, 2005.

[13] A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):28–40, 1993.

[14] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *CPC*, January 2000.

[15] H. Han and C.-W. Tseng. Improving locality for adaptive irregular scientific codes. In *LCPC*, pages 173–188, 2000.

[16] I. Kadayif and M. Kandemir. Data space-oriented tiling for enhancing locality. *Trans. on Embedded Computing Sys.*, 4(2):388–414, 2005.

[17] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. An integer linear programming approach for optimizing cache locality. In *Supercomputing*, pages 500–509, 1999.

[18] I. Kodukula and K. Pingali. Data-centric transformations for locality enhancement. *Int. J. Parallel Program.*, 29(3):319–364, 2001.

[19] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI, SIGPLAN Notices*, 2005.

[20] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[21] M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, April 1995.

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

[23] J. Mellor-Crummey and R. F. et al. HPCView: a tool for top-down analysis of node performance. *J. of Supercomputing*, 23:81–104, 2002.

[24] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, June 2001.

[25] B. Quaing, J. Tao, and W. Karl. Yaco: A user conducted visualization tool for supporting cache optimization. In *Proceedings of HPCC 2005*, volume 3726 of *Lecture Notes in Computer Science*, pages 694–703, 2005.

[26] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI, SIGPLAN Not.*, pages 215–228, 1999.

[27] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse Tiling for Stationary Iterative Methods. *International Journal of High Performance Computing Applications*, 18(1):95–113, 2004.

[28] E. Vanderdeijl, O. Temam, E. Granston, and G. Kanbier. The cache visualization tool. *IEEE Computer*, 30(7):71, 1997.

[29] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS*, volume 3038 of *LNCS*, pages 440–447, 2004.

[30] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI, SIGPLAN Notices*, pages 30–44, 1991.

[31] D. Wonnacott. Achieving scalable locality with time skewing. *Int. J. Parallel Program.*, 30(3):181–221, 2002.

[32] Y. Yu, K. Beyls, and E. H. D'Hollander. Visualizing the impact of cache on the program execution. In *Information Visualization 2001*, pages 336–341, 2001.

[33] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI, SIGPLAN Notices*, 2004.