The SLO Handbook

Kristof Beyls `<Kristof.Beyls@elis.UGent.be>`

The SLO Handbook

by Kristof Beyls

1.0.0

Abstract

SLO analyzes the causes of poor temporal data locality, and suggests program refactorings that are required to increase locality. After applying the suggested refactorings, the locality is improved, the number of data cache misses is typically reduced, and execution speed may be enhanced.

Chapter 1. Introduction

SLO helps you to identify the source code constructs that generate poor temporal data locality. Poor temporal data locality leads to many data cache misses and slow program execution. SLO aims to be a better cache analysis tool. Most other cache profilers indicate the source code lines that generate most cache misses. However, eliminating the cache misses often requires code changes (refactorings) in completely different statements, functions or even source files. In contrast, SLO highlights the loops and other code constructs that must be refactored to improve the data locality so that the cache misses are turned into cache hits.

Basically, SLO analyzes for each data reuse, which code is executed between use and reuse. Based on this analysis, it suggests an appropriate refactoring to improve the locality.

Please report any problems or feature requests to `<Kristof.Beyls@elis.UGent.be>`, or use the bug report and feature request features at the web-site http://www.sourceforge.net/projects/slo.

1. Comparison between using SLO and more traditional cache profilers

The operation of SLO is best illustrated by comparing it to other cache profiling tools. Using the example code below. The program below calculates inproducts and summations of all elements on a number of arrays.

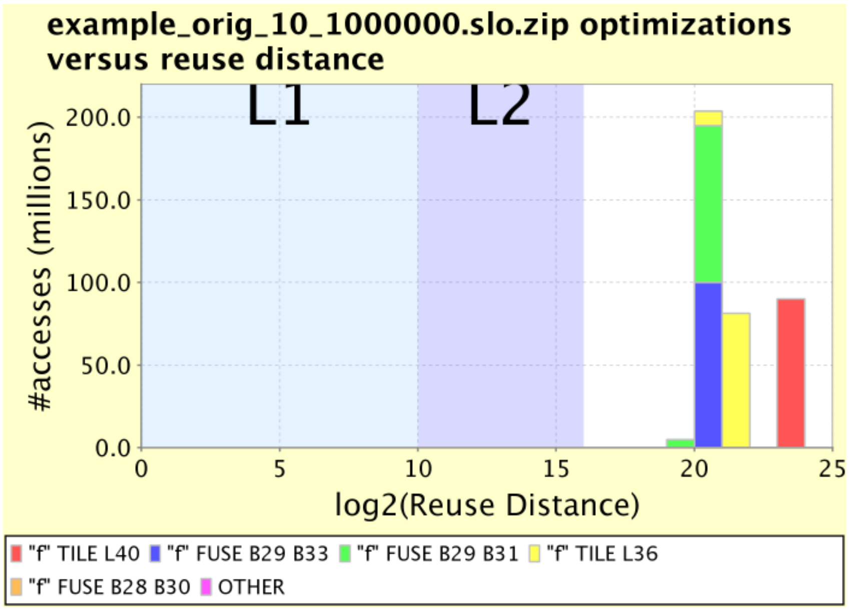
```
double inproduct (double *X, double *Y, int len)
{
   int i; double result=0.0;
   for(i=0; i<len; i++)
     result += X[i]*Y[i];
   return result;
}

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

void f
(double **X, double **Y, double** Z, int len, int N)
{
  int i,j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
      double inp = inproduct (X[i],Y[j],len);
      double sumX = sum (X[i],len);
      double sumY = sum (Y[j],len);
      Z[i][j] = inp+sumX+sumY;
    }
}
```

Most other cache profiling tools highlight the source code lines where most cache misses occur. For example, Valgrind reports the following source code lines where cache misses occur:

```
   L1     L2

     .      .  double inproduct (double *X, double *Y, int len)
     .      .  {
     .      .    int i; double result=0.0;
  0.0%   0.0%    for(i=0; i<len; i++)
 50.0%  50.0%      result += X[i]*Y[i];
     .      .    return result;
     .       .  }
     .       .
     .       .  double sum (double *X, int len)
     .       .  {
     .       .    int i; double result=0.0;
  0.0%    0.0%    for(i=0; i<len; i++)
 50.0%   50.0%      result += X[i];
     .       .    return result;
     .       .  }
     .       .
     .       .  void f
     .       .  (double **X, double **Y, double** Z, int len, int N)
     .       .  {
     .       .    int i,j;
  0.0%    0.0%    for (i=0; i<N; i++)
  0.0%    0.0%      for (j=0; j<N; j++) {
  0.0%    0.0%        double inp = inproduct (X[i],Y[j],len);
     .       .        double sumX = sum (X[i],len);
     .       .        double sumY = sum (Y[j],len);
  0.0%    0.0%        Z[i][j] = inp+sumX+sumY;
     .       .      }
     .       .  }
```

Valgrind shows that half of the cache misses occur insided the loop in `inproduct`; while the other half occur inside the loop in `sum`. While this information clearly shows where the misses occur, it is not directly obvious how to refactor the program so that these misses disappear.

In contrast, SLO analyzes cache misses in a more abstract way. Most cache misses occur on data that is reused, but for which the previous use is so far in the past, that the data has been evicted from the cache by other data that has been accessed since. SLO profiles the data reuses, and a histogram of reuses and their corresponding distance is generated, like the histogram below.

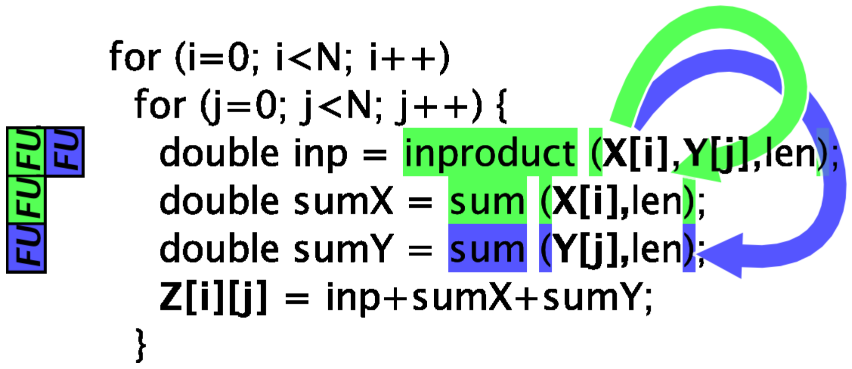


The histogram shows that for this particular run of the program, all reuses occur at distances larger than 2^{19}.

Note

The reuse distance estimates the minimum cache size needed for the data to remain in the cache between use and reuse. E.g., for a reuse with distance 2^{19}, the cache must be able to hold 2^{19} data elements. Therefore, the cache missing reuses can easily be read from the histogram as all reuses that are at a distance larger than the cache size.

The coloured background area, indicated with `L1` and `L2` show no reuses at those distances, meaning no reuses produce cache hits in either the L1 cache or the L2 cache. The different colors of the bars in the histogram indicate different refactorings that are needed for nearing the reuses. Bringing reuses closer together will move them to the left in the histogram, into the areas where L2 or even L1 cache hits occur. When you use the mouse and click on one of the colored bars, the corresponding refactoring is highlighted in the code.

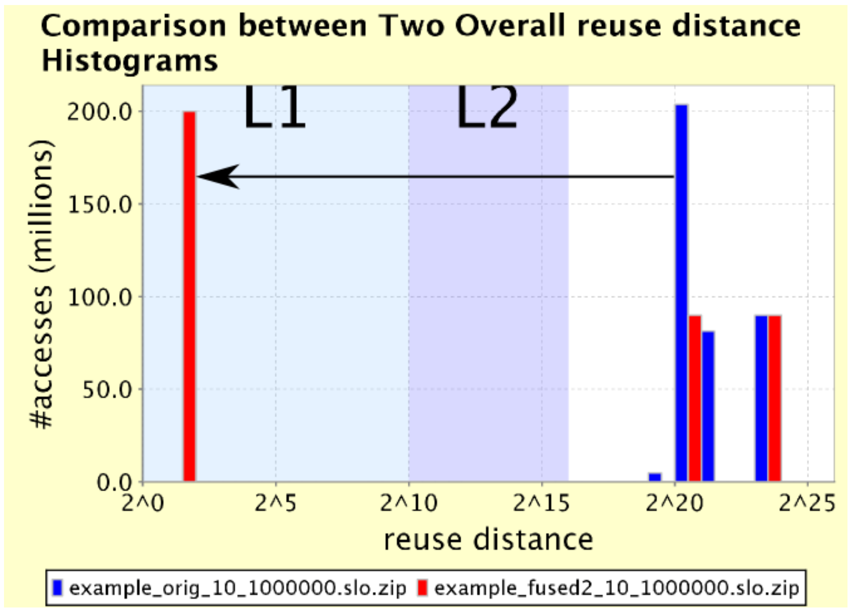


```
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    double inp = inproduct (X[i],Y[j],len);
    double sumX = sum (X[i],len);
    double sumY = sum (Y[j],len);
    Z[i][j] = inp+sumX+sumY;
  }
```

The screenshot shows that the blue reuses (at distance 2^{20} in the histogram) can be shortened by merging the computations in `inproduct` with the second call to `sum`. Similarly, the green reuses can be shortened by merging the computations of `inproduct` with the first call to `sum`. After merging the computations of in `inproduct` and `sum`, the code looks as follows:

```
void f
(double **X, double **Y, double** Z, int len, int N)
{
  int i,j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
      double inp=0.0, sumX=0.0, sumY=0.0;
      double *Xi=X[i], *Yj=Y[j];
      int i2;
      for(i2=0; i2<len; i2++) {
        inp += Xi[i2]*Yj[i2];
        sumX += Xi[i2];
        sumY += Yj[i2];
      }
      Z[i][j] = inp+sumX+sumY;
    }
}
```

SLO also allows to visually compare the histograms. In the screenshot below, the reuse distance histogram of the original code is shown in blue, while the reuse distance histogram of the optimized code is shown in red. As a result of the fusion, the large blue peak at distance 2^{20} has been moved to distance 2^{2}. Therefore, those reuses now generate L1 cache hits instead of L2 misses (see background colors). The resulting code runs about two times faster on a Pentium4.


Comparison between Two Overall reuse distance Histograms
L1
L2
200.0
150.0
100.0
50.0
0.0
#accesses (millions)
2^0
2^5
2^10
2^15
2^20
2^25
reuse distance
example_orig_10_1000000.slo.zip
example_fused2_10_1000000.slo.zip

Chapter 2. Using SLO

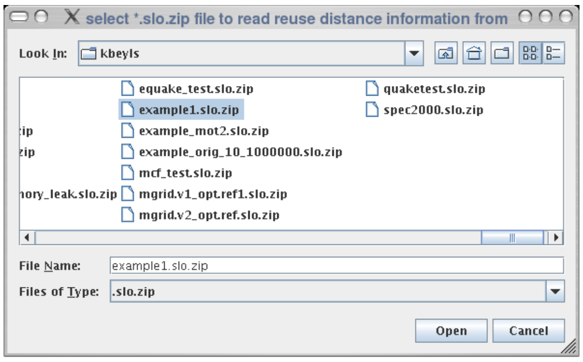
The input to SLO is a zip-file containing the source code of the program and the results of a profiling run. How to create this zip-file is documented in a separate chapter (Chapter 4, *Creating .slo.zip files*) , as it requires compiling your program with a special GCC-compiler. In this chapter, it is assumed that you already created such a .slo.zip file. Alternatively, you may download some of the example .slo.zip files from the SLO homepage to experiment with before profiling your own application.

1. Getting Started: How to start up SLO

1. There are a number of ways to start SLO:

From the command line	e.g. java -jar slo-1.0.jar example1.slo.zip.
From the web using web-start	By having Java webstart installed in your web browser and following the link http://slo.sourceforge.net/webstart/slo.jnlp. Using webstart ensures that you are always using the latest version available.

   If you don't specify a .slo.zip file on the command line, or when you start SLO using the webstart-link, a file selector pops up, as shown in the screenshot below. In this file selector, choose a .slo.zip file to continue. SLO cannot operate without a .slo.zip file!



select *.slo.zip file to read reuse distance information from

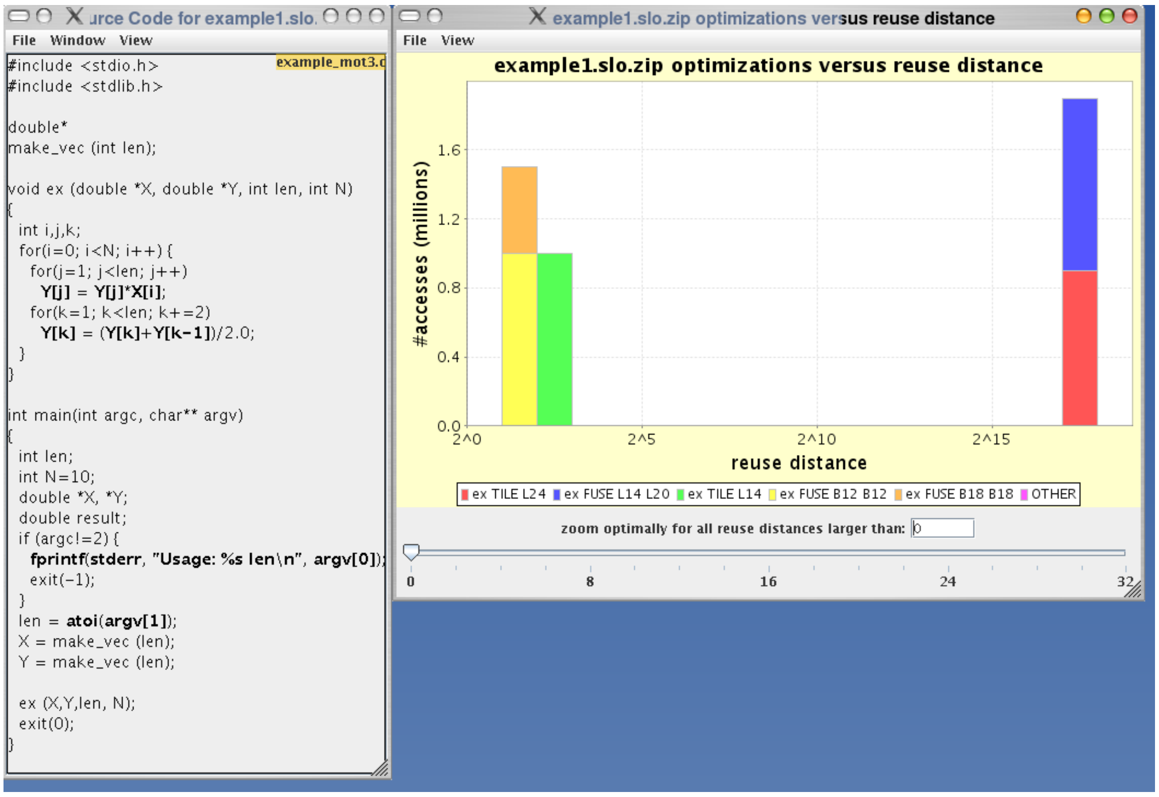
Look In: kbeyls

```
ip                   equake_test.slo.zip                   quaketest.slo.zip
zip                  example1.slo.zip                      spec2000.slo.zip
ory_leak.slo.zip     example_mot2.slo.zip
                     example_orig_10_1000000.slo.zip
                     mcf_test.slo.zip
                     mgrid.v1_opt.ref1.slo.zip
                     mgrid.v2_opt.ref.slo.zip
```

File Name: example1.slo.zip

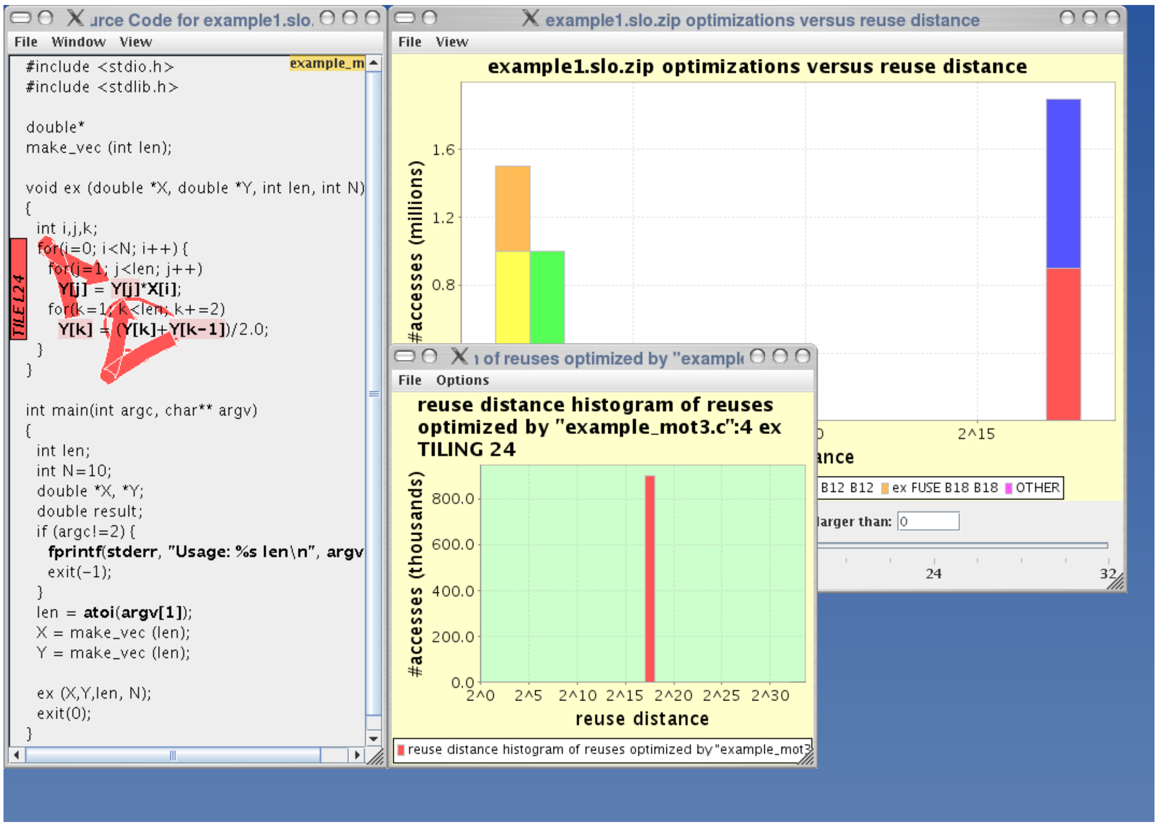
Files of Type: .slo.zip

2. After the .slo.zip file has been parsed (this can take a few seconds), two windows pop up: a source file window, and a histogram window containing the reuse distance histograms. For the input file `example1.slo.zip`, available from SLO's homepage, these windows look as follows:


urce Code for example1.slo.
File Window View
example_mot3.c
example1.slo.zip optimizations versus reuse distance
File View
example1.slo.zip optimizations versus reuse distance
#accesses (millions)
1.6
1.2
0.8
0.4
0.0
2^0
2^5
2^10
2^15
reuse distance
🟥 ex TILE L24 🟦 ex FUSE L14 L20 🟩 ex TILE L14 🟨 ex FUSE B12 B12 🟧 ex FUSE B18 B18 🟪 OTHER
zoom optimally for all reuse distances larger than: 0
0
8
16
24
32

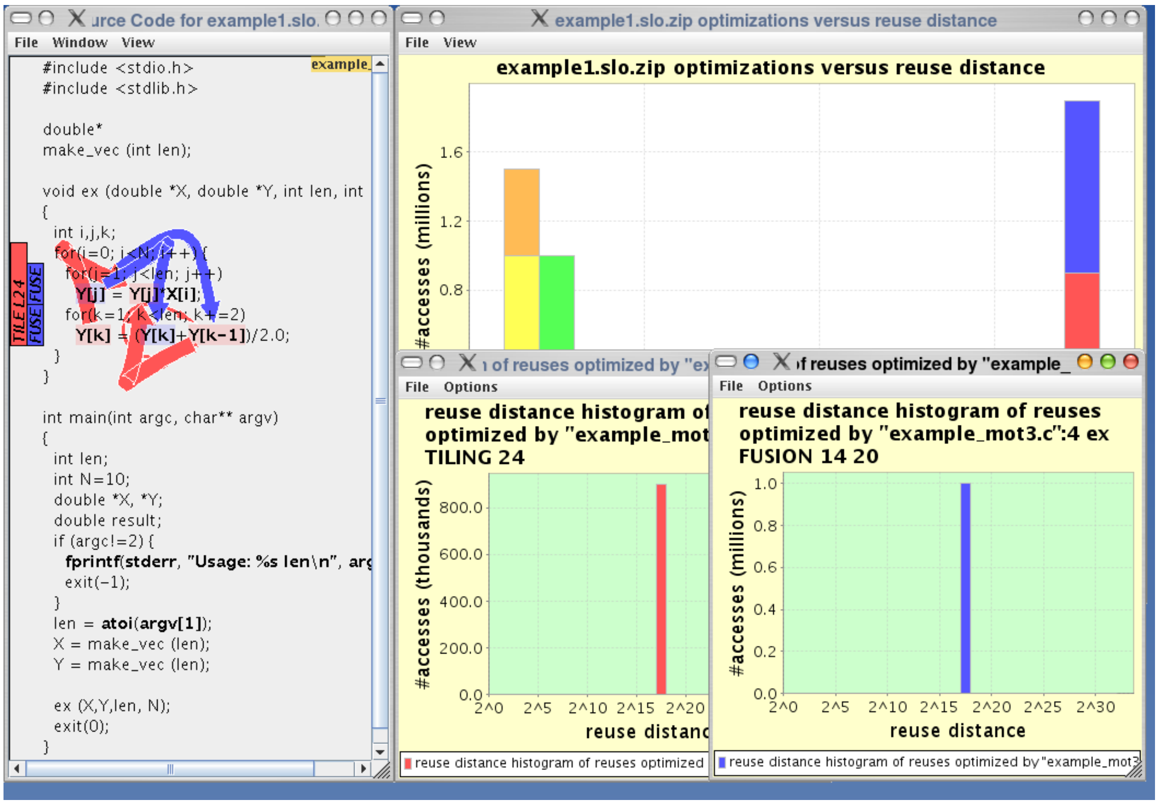
```
#include <stdio.h>
#include <stdlib.h>

double*
make_vec (int len);

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;
  }
}

int main(int argc, char** argv)
{
  int len;
  int N=10;
  double *X, *Y;
  double result;
  if (argc!=2) {
    fprintf(stderr, "Usage: %s len\n", argv[0]);
    exit(-1);
  }
  len = atoi(argv[1]);
  X = make_vec (len);
  Y = make_vec (len);

  ex (X,Y,len, N);
  exit(0);
}
```

3. Now you can start exploring the analysis results. The easiest way is to simply click on the colored bars in the histogram. Typically, you will start by exploring the long reuse distances. In the case of `example1.slo.zip`, the blue and the red reuse distances in the histogram at distance 2^{18}. Use your mouse and click on the red bar in the histogram. As a result, the corresponding tiling transformation (indicated as `ex TILE L24` in the legend), will be highlighted in the source code, and third window with a histogram will pop up. That third window contains the reuse distance histogram of all reuses that can be optimized by the `ex TILE L24`-optimization. It looks as follows:


urce Code for example1.slo.
File Window View
example_m
TILE L24
example1.slo.zip optimizations versus reuse distance
File View
example1.slo.zip optimizations versus reuse distance
#accesses (millions)
1.6
1.2
0.8
2^15
ance
B12 B12
ex FUSE B18 B18
OTHER
larger than: 0
24
32
n of reuses optimized by "exampl
File Options
reuse distance histogram of reuses optimized by "example_mot3.c":4 ex TILING 24
#accesses (thousands)
800.0
600.0
400.0
200.0
0.0
2^0 2^5 2^10 2^15 2^20 2^25 2^30
reuse distance
reuse distance histogram of reuses optimized by "example_mot3

```
#include <stdio.h>
#include <stdlib.h>

double*
make_vec (int len);

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;
  }
}

int main(int argc, char** argv)
{
  int len;
  int N=10;
  double *X, *Y;
  double result;
  if (argc!=2) {
    fprintf(stderr, "Usage: %s len\n", argv
    exit(-1);
  }
  len = atoi(argv[1]);
  X = make_vec (len);
  Y = make_vec (len);

  ex (X,Y,len, N);
  exit(0);
}
```

4. After checking the loop that needs to be tiled in the source code, and understanding why this is needed, you may also want to check the blue optimization. Clicking on the blue optimization will also highlight this optimization in the source code, and pop up a window with the reuse distance histogram of the reuses optimized by the blue optimization:


urce Code for example1.slo.
File Window View
example_
TILE L24
FUSE
FUSE
example1.slo.zip optimizations versus reuse distance
File View
example1.slo.zip optimizations versus reuse distance
#accesses (millions)
1.6
1.2
0.8
n of reuses optimized by "ex
File Options
reuse distance histogram of reuses optimized by "example_mot TILING 24
#accesses (thousands)
800.0
600.0
400.0
200.0
0.0
2^0 2^5 2^10 2^15 2^20
reuse distanc
reuse distance histogram of reuses optimized
of reuses optimized by "example_
File Options
reuse distance histogram of reuses optimized by "example_mot3.c":4 ex FUSION 14 20
#accesses (millions)
1.0
0.8
0.6
0.4
0.2
0.0
2^0 2^5 2^10 2^15 2^20 2^25 2^30
reuse distance
reuse distance histogram of reuses optimized by "example_mot3

```
#include <stdio.h>
#include <stdlib.h>

double*
make_vec (int len);

void ex (double *X, double *Y, int len, int
{
  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;
  }
}

int main(int argc, char** argv)
{
  int len;
  int N=10;
  double *X, *Y;
  double result;
  if (argc!=2) {
    fprintf(stderr, "Usage: %s len\n", arg
    exit(-1);
  }
  len = atoi(argv[1]);
  X = make_vec (len);
  Y = make_vec (len);

  ex (X,Y,len, N);
  exit(0);
}
```

Tip

You may clear all the currently highlighted optimizations by clicking the menu-item View → "Clear highlighted optimizations" in the main histogram window. This is helpful when too many optimizations are highlighted at once, and they clutter each other.

Chapter 3. Command Reference

1. The reuse distance histogram window

1.1. The File Menu

File → Export in SVG format	outputs the histogram in SVG (Scalable Vector Graphics) format. The file that the histogram is written to is selected through a file selector dialog that pops up.
File → Export in EPS format	outputs the histogram in an EPS (Encapsulated Postscript) format. The file that the histogram is written to is selected through a file selector dialog that pops up.
File → Export in PNG format	outputs the histogram in an PNG (Encapsulated Postscript) format. The file that the histogram is written to is selected through a file selector dialog that pops up. The file selector allows to set the resolution and width and height of the exported PNG drawing of the histogram.
File → Output Analysis data to CSV files	outputs the reuse distance data to a comma separated value (CSV) to a number of CSV-files in a newly created directory. The directory to store the CSV-files in is selected through a file selector dialog that pops up. An example of such a CSV file for the `example1.slo.zip` file is shown in Table 3.1, “Example of exported CSV file: example1.slo.zip.noncumul.absolute.csv”. It contains the histogram in table form. Eight of these CSV file are written, one for each possible combination of with or without "absolute", "noncumul" and "summary". "absolute" corresponds to a CSV file with absolute values for the number of reuses at a certain distance for a certain refactoring. "relative", or non-"absolute", corresponds to a CSV-file where the relative number of reuses at a certain distance are encoded. This corresponds to the histogram after clicking View "Show Relative Histogram". For each distance, the numbers add up to 1. "noncumul" contains the regular histograms, while the "cumul"-versions contain the reverse-cumulative histograms, corresponding to clicking View "Reverse Cumulative Histogram". In the "summary" CSV-files, respectively all tiling-like, fusion-like, and function-fusion-like optimizations are combined.
File → Compare with other ...	Allows to compare the reuse distance histogram with the histogram in another .slo.zip file. After clicking this menu option, a file selector pops up to select another .slo.zip file. After selecting, the histograms from both .slo.zip files are shown in a single chart. This comes in handy, e.g., to see in what way the reuse distance histogram has changed after a refactoring. An example is shown in Figure 3.1, “Screenshot of comparison between two reuse distance histograms”

Figure 3.1. Screenshot of comparison between two reuse distance histograms

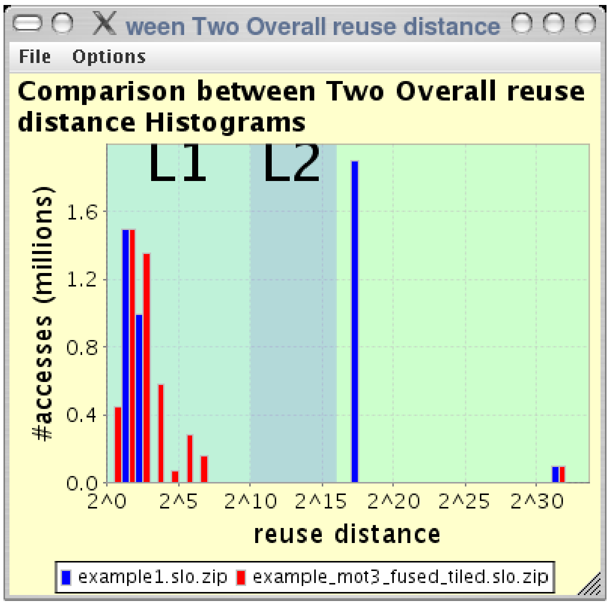


Table 3.1. Example of exported CSV file: example1.slo.zip.noncumul.absolute.csv

reuse_distance	ex_TILE_L24	ex_FUSE_L14_L20	ex_TILE_L14	ex_FUSE_B12_B12	ex_FUSE_B18_B18
0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	999990.0	500000.0
2.0	0.0	0.0	999980.0	0.0	0.0
3.0	0.0	0.0	0.0	0.0	0.0
4.0	0.0	0.0	0.0	0.0	0.0
5.0	0.0	0.0	0.0	0.0	0.0
6.0	0.0	0.0	0.0	0.0	0.0
7.0	0.0	0.0	0.0	0.0	0.0
8.0	0.0	0.0	0.0	0.0	0.0
9.0	0.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	0.0	0.0
11.0	0.0	0.0	0.0	0.0	0.0
12.0	0.0	0.0	0.0	0.0	0.0
13.0	0.0	0.0	0.0	0.0	0.0
14.0	0.0	0.0	0.0	0.0	0.0
15.0	0.0	0.0	0.0	0.0	0.0
16.0	0.0	0.0	0.0	0.0	0.0
17.0	899991.0	999990.0	0.0	0.0	0.0
18.0	9.0	0.0	0.0	0.0	0.0
19.0	0.0	0.0	0.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0
21.0	0.0	0.0	0.0	0.0	0.0
22.0	0.0	0.0	0.0	0.0	0.0
23.0	0.0	0.0	0.0	0.0	0.0
24.0	0.0	0.0	0.0	0.0	0.0
25.0	0.0	0.0	0.0	0.0	0.0
26.0	0.0	0.0	0.0	0.0	0.0
27.0	0.0	0.0	0.0	0.0	0.0
28.0	0.0	0.0	0.0	0.0	0.0
29.0	0.0	0.0	0.0	0.0	0.0
30.0	0.0	0.0	0.0	0.0	0.0
31.0	0.0	0.0	0.0	0.0	0.0
32.0	0.0	0.0	0.0	0.0	0.0

1.2. The View Menu

View → Highlight 10 most important optimizations	Highlights the 10 most important optimizations in the source code, as if the user had individually clicked on the 10 most important optimizations in the histogram. The most important optimizations are determined by computing a weight for each optimization. The larger the weight, the more important the optimization is considered to be. The formula to compute this weight is:

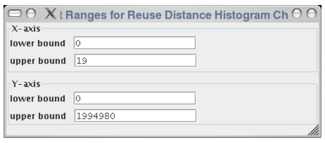
$$\text{weight}(p) = \sum_{l} l \times \#\{r|r \text{ is optimized by } p \text{ and } 2^{l-}$$

View → Clear highlighted optimizations

Deletes all indications in the source code of the optimizations that have been highlighted so far.

View → Set chart ranges

Pops up the following dialog that lets you set the lower and upper bounds of the X and Y axes in the histogram.



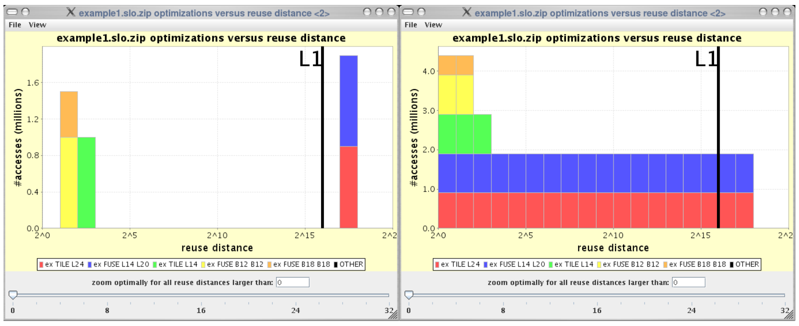
Ranges for Reuse Distance Histogram Ch

X-axis
lower bound: 0
upper bound: 19

Y-axis
lower bound: 0
upper bound: 1994980

View → Reverse cumulative histogram

When this option is selected, the reverse cumulative histogram is shown, instead of a regular histogram. In a reverse cumulative histogram, all reuses that occur at larger distances are represented in each bar, instead of only the reuses with the same distance. In this view, all misses for a specific cache size can more easily be determined. E.g. consider the figures below show both the normal(left) and the reverse cumulative histograms(right) of a certain program run. The size of the L1 cache is indicated by a vertical black line. To find out which reuses miss the cache in the left, normal, histogram, you need to look at all bars to the right of the L1 cache size. In contrast, to find out the cache missing reuses fromthe reverse cumulative histogram (right), you just need to look to the reuses that are displayed where the L1 cache size is drawn. (TODO: look for a screenshot where the advantage of using a reverse cumulative histogram is more obvious)


example1.slo.zip optimizations versus reuse distance <2>
File View
example1.slo.zip optimizations versus reuse distance
L1
#accesses (millions)
reuse distance
2^0
2^5
2^10
2^15
2^2
🟥 ex TILE L24 🟦 ex FUSE L14 L20 🟩 ex TILE L14 🟨 ex FUSE B12 B12 🟧 ex FUSE B18 B18 ■ OTHER
zoom optimally for all reuse distances larger than: 0
0
8
16
24
32

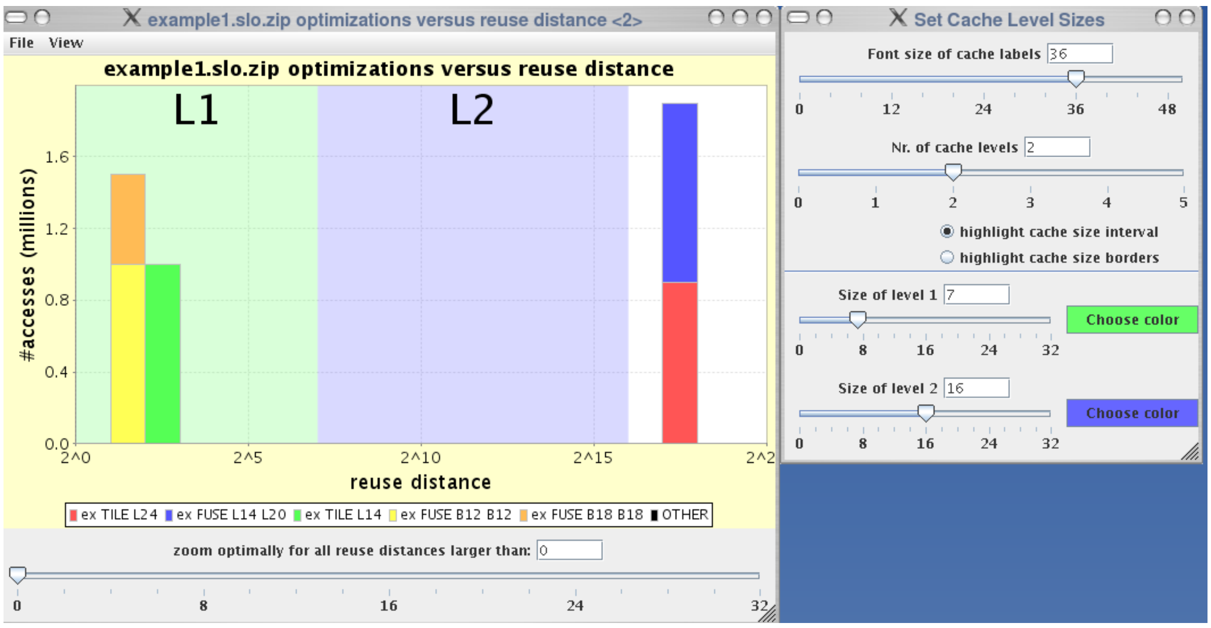
View → Configure cache sizes

This option is used to set the cache sizes that are drawn in the background of the histogram view. An example of two different modes of cache size indication is shown in Figure 3.2, “Screenshot of two possible indications of cache sizes in the reuse distance histogram: by interval and by borders”. After clicking, a dialog pops up to set the cache sizes, an example of which is also found in Figure 3.2, “Screenshot of two possible indications of cache sizes in the reuse distance histogram: by interval and by borders”. From top to bottom, the dialog contains following controls:

Font size of cache labels	controls the size in pixels of the "L1", "L2" etc. labels of the cache size in the histogram
Nr. of cache levels	controls the number of cache levels indicated
Highlight cache size interval/border	controls whether cache sizes are visualized by a colored background

		(interval, see first screenshot above), or by vertical lines at the cache size boundaries (borders, see second screenshot above).
	Size of Level x	For each of the number of cache levels selected above, a slider is used to determine the size of the cache level. The size is expressed as the log2 of the reuse distance. On the right, the 'Choose color' button allows to select the color to be used for the background, when cache sizes are visualized by a colored background (see interval mode above).
View → Write cache sizes to file	pops up a file selector to save the cache size info to a file. Make sure to end the name of the files with '.cachesizes'	
View → Read cache sizes from file	pops up a file selector to load an earlier saved the cache size info.	

Figure 3.2. Screenshot of two possible indications of cache sizes in the reuse distance histogram: by interval and by borders



zoom optimally for all reuse distances larger than: 0

Font size of cache labels 36

Nr. of cache levels 2

[✓ highlight cache size interval]

highlight cache size borders

Size of level 1 7

Size of level 2 16

2. The source code window

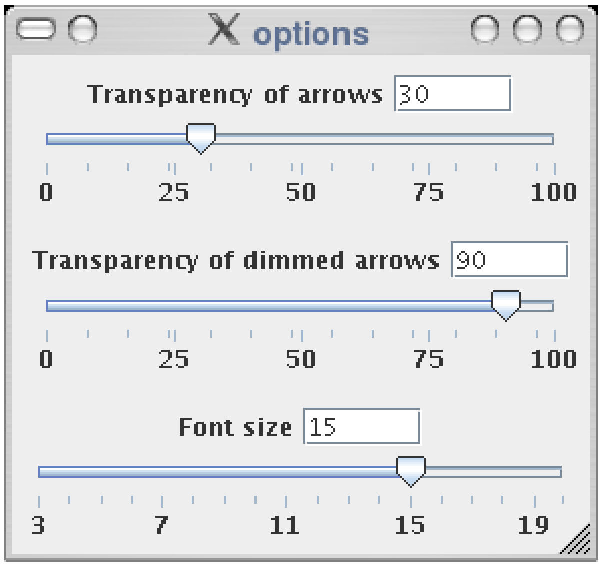
2.1. The File Menu

File → Export complete in SVG format	outputs the contents of the source code window in SVG (Scalable Vector Graphics) format. Also the non-visible part of the window is exported.
File → Export visible in SVG format	outputs the contents of the source code window in SVG (Scalable Vector Graphics) format. Only the visible part of the window is exported.
File → Export in EPS format	outputs the contents of the source code window in EPS (Encapsulated Postscript) format.

File → Exit	exits SLO.

2.2. The Window Menu

Window → Options Window	shows up a dialog to set preferences about the transparency of arrows and the font size of the source text. (see screenshot ???)



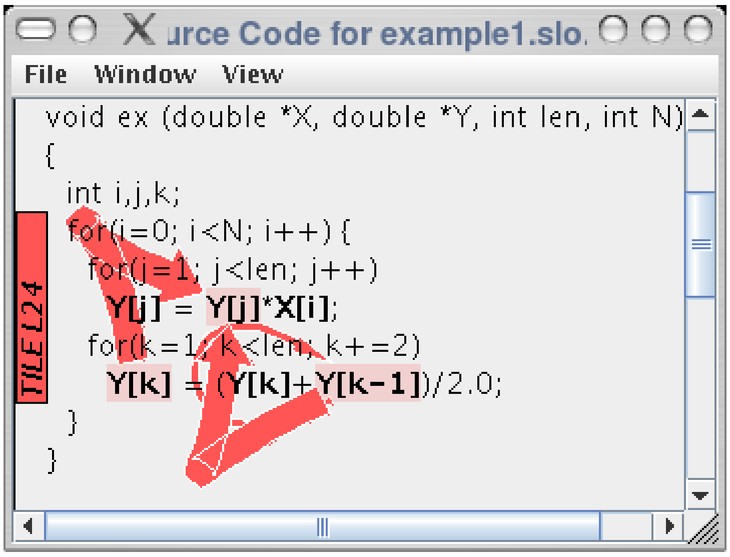
Transparency of arrows 30

Transparency of dimmed arrows 90

Font size 15

2.3. The View Menu

View → Transparent arrows	controls whether arrows are transparent or not.
View → Show linenumber	show linenumbers in the source code
View → Arrow thickness follows its weight.	when unselected, all arrows have the same width. When selected, the width (thickness) of the arrows depend on the number of reuses they represent. The thickness is proportional to the percentage of reuses that the arrows represents of the number of reuses that are optimized by the refactoring it belongs to. (For each suggested optimization/refactoring, multiple arrows may be present). An example is given in screenshot.
View → Decrease minimum arrow length	decreases the minimum length of an arrow. This is mostly important for visualizing arrows for which the source and sink is the same source code.
View → Increase minimum arrow length	increases the minimum length of an arrow.
View → Zoom xxx	for each source code file, there's a menu item with 'Zoom xxx', where xxx is the source code file name. When the menu option is selected, that file is zoomed, i.e. the source code is drawn (the font size can be selected using the options window, see screenshot ???). When the menu item is unselected, each line in the source code file is represented by a line that is only a pixel in height.



```
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;
  }
}
```

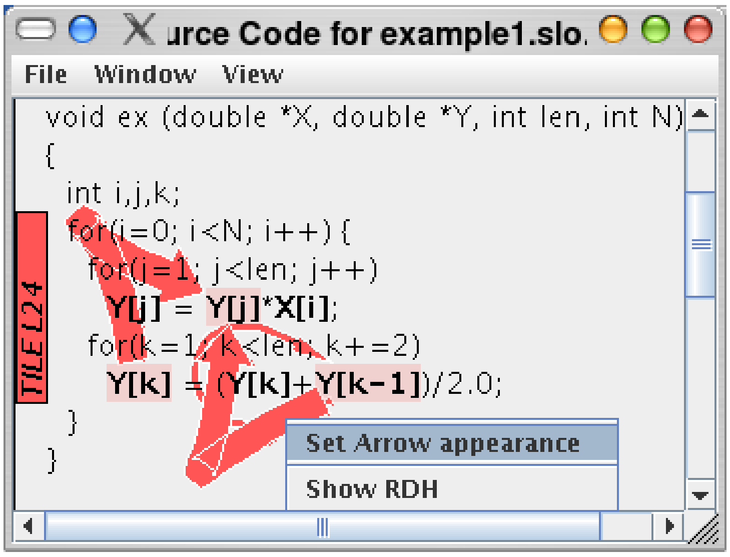
2.4. Other Interactive Actions

2.4.1. Interactive Actions in the Histogram window

In the histogram window, when clicking on a bar of a specific color, the corresponding optimization is highlighted in the source code window.

2.4.2. Interactive Actions in the Source Code window

When clicking an arrow with the mouse, a menu with two options appears: see screenshot???. The first option, "Set arrow appearance" pops up a dialog box, that allows to set the angle in which the arrow starts and ends. It also allows to set the length of the arrow. The second option, "Show RDH", pops up a window with the histogram of reuse distances of the reuses represented by that arrow.



```
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;
  }
}
```

3. Command line options

`java -jar slo.jar` [[-c] | [--cut-off-percentage]*`percentage`*] [--default-cache-sizes-file *`filename.cachesize`*] [--x-lowerbound *`x0`*] [--y-lowerbound *`y0`*] [--x-upperbound *`x1`*] [--y-upperbound *`y1`*]
[--dump-histogram-to-png-file *`filename.png`*] [--png-resolution *`resolution`*]
[--png-width *`width`*] [--png-height *`height`*]
[--dump-to-html *`dirname`*] [--max-nr-series-drawn *`nrSeries`*]

[--read-preferences *filename.xml*] [--store-preferences *filename.xml*]
[--dump-nr-refactorings-needed-vs-cachesize *percentage*] *filename.slo.zip*

`-c` *percentage*	This option specifies that only *percentage* percent of all reuses need to be read. The less important reuse information doesn't need to be read in. This can be useful to reduce the memory usage of SLO.
`--cut-off-percentage` *percentage*	See option `-c`
`--default-cache-sizes-file` *filename.cachesize*	At start-up, read in cache size info from `filename.cachesize` file *filename.cachesize*.
`--x-lowerbound` *x0*	Set default lower bound for X-axis in histogram to *log2(x0)*.
`--y-lowerbound` *y0*	Set default lower bound for Y-axis in histogram to *y0*.
`--x-upperbound` *x1*	Set default upper bound for X-axis in histogram to *log2(x1)*.
`--y-upperbound` *y1*	Set default upper bound for Y-axis in histogram to *y1*.
`--dump-histogram-to-png-file` *filename.png*	Take a screenshot of the reuse distance histogram, and dump it in PNG file format to file *filename.png*. After dumping the histogram, SLO exits.
`--png-resolution` *resolution*	Set the resolution of the png-file dumped by option `--dump-histogram-to-png-file` to *resolution* dpi.
`--png-width` *width*	Set the width of the png-file dumped by option `--dump-histogram-to-png-file` to *width* inches.
`--png-height` *height*	Set the height of the png-file dumped by option `--dump-histogram-to-png-file` to *height* inches.
`--dump-to-html` *dirname*	Create a web-page with the reuse distance histogram. In the web-page, when a bar is clicked in the reuse distance histogram, and JavaScript is enabled in the browser, the corresponding suggested refactoring is highlighted in a second browser frame. All files are dumped to directory *dirname*. The browse the result with a web-browser, browse to file `index.html` in directory *dirname*. After generating the HTML pages, SLO exits.
`--max-nr-series-drawn` *nrSeries*	The number of refactorings suggested by SLO may grow large for some complex programs. As a result, the reuse distance histogram contains many different colored bars. This may slow down the interactive drawing of the histogram too much. Therefore, this option allows to limit the number of colored bars drawn to *nrSeries*, so that SLO reacts quicker to user interaction. By default, the value of `nrSeries` is 40. All other less-important refactorings are combined into a single category named `"other"`, and are drawn in black.
`--read-preferences` *filename.xml*	Reads preferences from the XML-file *preferences.xml*. For the list of preferences that can be set: see Section 4, “Preferences file options”
`--store-preferences` *filename.xml*	Writes preferences to the XML-file *preferences.xml*. The file is written after exiting SLO. For the list of preferences that can be set: see Section 4, “Preferences file options”
`--dump-nr-refactorings-needed-vs-cachesize` *percentage*	Writes out to standard output the minimum number of refactorings that must be applied to optimize at least *percentage* percent of all reuse distance longer than a given cache size. After dumping the table to stdout, SLO exits.

4. Preferences file options

Some controlling variables that are not easily specified on the command line are stored as "user preferences". This means, that the settings controlled by the variables below remain persistent between different invocations of the SLO tool. Furthermore, these options can be altered by the command line options `--read-preferences`. The easiest way to change the options, if they are not settable from some of the menu's in SLO, or from another command line option, is to first run SLO with the `--store-preferences` command line option. This dumps all options to an XML-file. Then, change the value of the variables that you want to change. Subsequently, run SLO again with `--read-preferences`, so that the preferences are read from the altered XML file. This will result in the preferences read to become persistent for all following runs of SLO, until you specifically change these preferences again. An overview of preference variables is shown in Table 3.2, “Overview of preference variables”.

Note

Most of the options below should become changeable from menu options or from the command line in future versions of SLO. Currently, most of them are only settable through the `--read-preferences` option since this can be implemented more quickly than adding menu options. When it is determined that the controlling variables are truely useful, they will somehow be settable from menu options are command line options.

Table 3.2. Overview of preference variables

Preference variable	**Meaning**
`CacheSizesValueMarkerStrokeThickness`	Specifies the thickness of cache size markers in the reuse distance histogram, when the cache sizes are indicated by lines, see screenshot Figure 3.2, “Screenshot of two possible indications of cache sizes in the reuse distance histogram: by interval and by borders”. The thickness is indicated in number of pixels.
`DisplayHistogramVSReuseDistanceChartTitle`	Boolean value, controlling whether the title in the reuse distance histogram window should be displayed. When set to true, the title displays the name of the .slo.zip file and the distance metric (reuse distance or reference distance)
`OptimizationsVsReuseDistancesPlotNrSeriesVisibleInLegend`	controls the maximum number of series drawn in the reuse distance histogram. Also see command line option `--max-nr-series-drawn`
`HTMLOverallHistogramWidth`	controls the width (in number of pixels) of the histograms generated in the HTML output. Also see command line option `--dump-to-html`
`HTMLOverallHistogramHeight`	controls the height (in number of pixels) of the histograms generated in the HTML output. Also see command line option `--dump-to-html`

Chapter 4. Creating .slo.zip files

1. Overview

The input to SLO is a zip-file, ending with extension .slo.zip. This zip-file is created by instrumenting the program that you want to optimize using the GCC-SLO compiler with option `-fslo-instrument`. During compilation a number of files are generated that describes the source locations of memory accesses, basic blocks, functions and their control flow graphs. After running the instrumented program, a file with name `BRD` will be created. This file contains all recorded run-time reuse distance information. These files then need to be combined in a zip file. An overview of the different kinds of files in the .slo.zip file is given below:

source code files	The source code files contain the source code of your program, so that SLO can show suggestions in your source code.
.bb_info files	For each compiled file source.c, the .slo.zip file contains a source.c.bb_info file. This file contains information generated by the GCC-SLO compiler about the source code locations of all the basic blocks in the program. The syntax of these files is described in Section 4.2, “Syntax of the .bb_info files”.
.function_info files	For each compiled file source.c, the .slo.zip file contains a source.c.function_info file. This file contains information generated by the GCC-SLO compiler about the functions in the source file, the basic blocks they contain and their control flow graphs. The syntax of these files is described in Section 4.3, “Syntax of the .function_info files”.
.memaccess_info files	For each compiled file source.c, the .slo.zip file contains a source.c.memaccess_info file. This file contains information generated by the GCC-SLO compiler about the source locations of the memory references in the source file. The syntax of these files is described in Section 4.4, “Syntax of the .memaccess_info files”.
BRD file	This file contains all reuse distance histogram information needed by SLO to compute the reuse distance histograms, associated refactorings and associated arrows. This BRD file is created by running the program that is instrumented by the GCC-SLO compiler. The syntax of this file is described in Section 4.1, “Syntax of the BRD file”.

2. Installing the GCC-SLO compiler

The current GCC-SLO compiler is based on the GNU GCC compiler version 4.1. It has been extended to recognize option `-fslo-instrument`. When this option is used during compilation, all necessary compile-time information for SLO is dumped to .*_info files, and the resulting binary code is instrumented so that all necessary run-time information about reuses and their distance is recorded to a `BRD` file. The first step in analyzing your own programs with SLO is installing the GCC-SLO compiler, so that you can instrument your own program with it. The GCC-SLO compiler can be downloaded from following the download link at http://www.sourceforge.net/projects/slo. At the time of writing, the latest release is `gcc-slo-1.0.1-4.1.0.tar.gz`.

Currently, no binaries are distributed: the GCC-SLO compiler is only distributed as source code. To build the compiler, you need the same software and libraries installed, as you would need to build the GCC compiler for C, C++ and Fortran. The prerequisites for building GCC are listed at http://gcc.gnu.org/install/prerequisites.html. Once you are sure that the prerequisites are met, the compiler can easily be built by running the script `build-gcc-slo.sh` included in `gcc-slo-1.0.1-4.1.0.tar.gz`. The script automatically builds the compiler for you, and installs it in `$HOME/gcc-slo`.

3. Instrumenting programs using GCC-SLO

How to instrument your own programs with GCC-SLO is easiest explained by giving an example. Below, we'll show how to create `example1.slo.zip`, the example that is used for most of the screenshots in this manual. If you would like to follow these steps on your own computer, download `example1.tar.gz` at url http://slo.sourceforge.net/examples/example1.tar.gz, and unpack it. The file `example1.slo.zip` will be build by running GNU **make**. The contents of the `Makefile` is as follows:

```
CC=$(HOME)/gcc-slo/bin/gcc-slo
CCC=$(HOME)/gcc-slo/bin/g++-slo
INSTR_FLAG= -O3 -fno-inline -fslo-instrument
INSTR_LINK_OPTIONS= -static -lrd

all: example1 example1.slo.zip

clean:
        rm *.slo_instr_o *.o example1 example1_slo_instr *_info BRD example1.slo.zip

%.o: %.c
        $(CC) -O2 -c -o $@ $<

%.slo_instr_o: %.c
        $(CC) $(INSTR_FLAG) -c -o $@ $<

example1: example1.o make_vec.o
        $(CC) -o $@ example1.o make_vec.o -lm

example1_slo_instr: example1.slo_instr_o make_vec.o
        $(CCC) -o $@ example1.slo_instr_o make_vec.o $(INSTR_LINK_OPTIONS) -lm

BRD: example1_slo_instr
        ./example1_slo_instr 1000

example1.slo.zip: BRD
        zip $@ BRD example1.c example1.c.bb_info example1.c.function_info \
               example1.c.memaccess_info
```

Below, the lines in the `Makefile` are explained one by one. Following this explanation shows how the `example1.slo.zip` is built step by step:

`CC=$(HOME)/gcc-slo/bin/gcc-slo`	This sets variable `CC` to point to the gcc-slo C compiler.
`CCC=$(HOME)/gcc-slo/bin/g++-slo`	This sets variable `CCC` to point to the g++-slo C++ compiler.
`INSTR_FLAG= -O3 -fno-inline -fslo-instrument`	This sets `INSTR_FLAG` to contain the optimization flags during instrumenting compilation. The flag `-fslo-instrument` has the effect that all memory accesses, function entries and exits and basic block transitions will be instrumented appropriately. Furthermore, at compile time, for each source code file, the .bb_info, .function_info and .memaccess_info will be generated. The options `-O3` is the regular `-O3` option of GCC. The option `-fno-inline` is necessary, since otherwise the instrumenter will not see function boundaries of functions that were inlined.
`INSTR_LINK_OPTIONS= -static -lrd`	This sets variable `INSTR_LINK_OPTIONS` to contain the necessary link time options for profiling applications. The option `-lrd` links library `librd` which contains the necessary code to measure reuses, their distances and all other necessary run-time measurement. Option `-static` makes sure that library `librd` is statically linked in the final executable.
`%.o: %.c`	Describes the default way to compile a .c-file into a .o-file, without instrumenting.

`%.slo_instr_o: %.c`	Describes how a .c-file can be compiled with the necessary instrumentation. Instead of generating a .o file, the filename ends in .slo_instr_o.
`example1: example1.o make_vec.o`	Describes the default way to create executable example1, without instrumentation
`example1_slo_instr: example1.slo_instr_o make_vec.o`	Describes how to compile `example1_slo_instr`, where all code in `example1.c` is instrumented. Note that the code in `make_vec.c` is not instrumented, since `make_vec.o` is used instead of `make_vec.slo_instr_o`. Usually, it is best to instrument all your source code files.
`BRD: example1_slo_instr`	The `BRD` file contains all reuse information as measured by profiling the instrumented program.
`example1.slo.zip: BRD`	Describes that the `example1.slo.zip` file must contain the `BRD` file, all source code files and all *_info files generated at compile time.

3.1. Environment variables

A number of environment variables influence the profiling process. To influence the characteristics that are measured, set the following environment variables before running the instrumented program.

RDLIB_MEASURE_TRUE_REUSE_DISTANCE	If this environment variable is set to a value different from 0, the distance between reuses is measured in terms of "reuse distance", i.e. the number of different data elements accessed between both reuses. Otherwise, the distance is measured by "reference distance" a.k.a. "time distance", which is the total number of accesses between both reuses. The measurement of "reference distance" is much faster than the measurement of "reuse distance". On the other hand, "reuse distance" has a clearer connection with cache behavior.
RDLIB_EXPORT_TRACE	If this variable is set, during profiling the trace of memory accesses will be saved to a file. The name of the file is the value that this variable is set to. The trace file is a binary file, consisting of a sequence of memory accesses. For each memory access, the following bytes are written to the trace. First, the address of the accessed address is written, which is `sizeof(void*)` bytes long. Next, an identifier for the instruction generating the memory access is written, which is `sizeof(int)` bytes long.
RDLIB_EXPORT_TRACE_FLUSH_AFTER_EACH_ACCESS	If this variable is set to a value different from 0, and `RDLIB_EXPORT_TRACE` is set, the memory access trace will be flushed to disk after each memory access. This slows down the tracing considerably, but can be handy for detecting bugs.

4. Syntax of the files generated by GCC-SLO

4.1. Syntax of the BRD file

TODO

4.2. Syntax of the .bb_info files

TODO

4.3. Syntax of the .function_info files

The .function_info file represent the control flow graphs of all functions encountered by gcc-slo while compiling a single source file. An example .function_info file is shown in Example 4.1, “.function_info file contents example”. The formal syntax is described as follows, with the meaning in comments:

`function-info-file`	`function-info`* // a function-info file contains a list of function-info's, which each contains info about a single function
`function-info`	`( string basic-block-info* )` // function-info describes a single function. The string contains the function name. Then a list of basic-block-info's follows describing the different basic blocks and how they are linked in the control flow graph.
`string`	`"` [^"]* `"` // A string is any sequence of characters between double quotes
`basic-block-info`	`( natural = hexnumber hexnumber* )` // basic-block info encodes a single basic block in the function, and shows which basic blocks are successors in the control flow graph. The encoding is a little bit special: The natural represents the basic block number. In each function, the basic block are typically numbered from 0. This number corresponds with the basic block numbers found in .bb_info and BRD files. In the .function_info file, each basic block is also identified by a hexadecimal number. The first hexadecimal number, i.e., the one after the equals sign, is the hex-number for this basic block. The following hexadecimal numbers are the hexnumbers for the successors in the control flow graph.
`natural`	[0-9]+
`hexnumber`	`(0x` | `0X`)? [0-9|a-f|A-F]+

Example 4.1. .function_info file contents example

```
(1 "gnu_dev_major" (0=0xb7d6c050 0xb7d6c0f0 ) (2=0xb7d6c0f0 ) )
(2 "gnu_dev_minor" (0=0xb7d6c230 0xb7d6c2d0 ) (4=0xb7d6c2d0 ) )
(3 "gnu_dev_makedev" (0=0xb7d6c460 0xb7d6c500 ) (6=0xb7d6c500 ) )
(4 "ex" (0=0xb7d89b90 0xb7d89c30 ) (8=0xb7d89c30 0xb7d89eb0 ) (10=0xb7d89c80 0xb7d89d20 )
  (12=0xb7d89cd0 0xb7d89d20 ) (14=0xb7d89d20 0xb7d89cd0 0xb7d89d70 ) (16=0xb7d89d70 0xb7d89e10 )
  (18=0xb7d89dc0 0xb7d89e10 ) (20=0xb7d89e10 0xb7d89dc0 0xb7d89e60 ) (22=0xb7d89e60 0xb7d89eb0 )
  (24=0xb7d89eb0 0xb7d89c80 0xb7d89f00 ) (26=0xb7d89f00 ) )
(5 "main" (0=0xb7d96410 0xb7d964b0 ) (28=0xb7d964b0 0xb7d96500 0xb7d96550 ) (30=0xb7d96500 0xb7d8ee6c )
  (31=0xb7d8ee6c 0xb7d8eea0 ) (33=0xb7d8eea0 ) (34=0xb7d96550 0xb7d90b40 ) (35=0xb7d90b40 0xb7b595c8 )
  (37=0xb7b595c8 0xb7b594b0 ) (38=0xb7b594b0 0xb7b595f0 ) (39=0xb7b595f0 0xb7b594d8 ) (40=0xb7b594d8 0xb7d8efa4 )
  (41=0xb7d8efa4 0xb7d93444 ) (43=0xb7d93444 ) )
```

4.4. Syntax of the .memaccess_info files

TODO

Chapter 5. Example of Using SLO to optimize some SPEC2000 programs

In this chapter, a few examples will be shown on how SLO was used to optimize the temporal data locality in a number of the programs in the SPEC2000 benchmark.

1. Optimizing 179.art

TODO

2. Optimizing 181.equake

TODO

3. Optimizing 175.vpr

TODO

Chapter 6. Questions and Answers

TODO

Chapter 7. Credits and License

SLO

Contributors: