Analysis of Memory Behavior of DIS Stressmark Suite and Optimization

Haitao Du

November 20, 2000

Advisor: Rajesh Gupta
Center for Embedded Computer Systems
University of California, Irvine

Contents

List of Figures

1 Introduction

In this report, the access pattern of DIS Stressmark suite is analyzed in detail. The focus is on what is the major source of cache misses and how to optimize it. The appropriate optimization mechanisms are proposed. Their effectiveness is verified by the simulation on the *Simscalar*.

2 Baseline Architecture

The baseline architecture used throughout our analysis is SGI architecture. The corresponding cache organization is shown in the table below.

Cache	Size	Block_Size	Assoc
D1	32KB	32B	2-Way
I1	32KB	32B	2-Way
D2/I2(Unified)	1MB	64B	2-Way

Table: Baseline SGI Architecture

3 Analysis of Memory Access Behavior Access of DIS Stressmark Suite

When each of the 6 DIS Stressmarks is analyzed, the following points are our focuses:

- What is the major source of memory accesses (both load and store).
- What is the major source of L1/L2 data cache misses.
- What are the possible ways to optimize cache.

Several optimization techniques were referred to in this report. You can find more details about *Victim_Cache* and *Stream_Buffer* in [1], *Column_Cache* in [2], and *Blocking* (*Tiling*) in [3].

3.1 Pointer Stressmark

3.1.1 Stressmark Kernel and Source of Memory Accesses

In the kernel body of *PointerStressmark*, the entire process is performed multiple times for each run. Each time the execution is referred to as "thread". For each thread, multiple "hops" of searching is done for medians. In each hop, a window with size of w is scanned. Figure 1 shows the kernel of Pointer Stressmark.

Most of the memory accesses are generated during the process of searching for medians, which is in the range from (1) to (4). It can be seen that each time when searching for a median,

```
...
for (l=0; l<#thread; l++){
   ...
index = thread[l].initial;
while (hops<#hops){
   ...
      partition = field[index];
      for (ll=0; ll<w; ll++){      [1]
        ...
          x = field[index+ll];
          if (x>max) high++;       [2]
          else if (x>min){         [3]

            partition = x;
            balance = 0;

            for (lll=ll+1; lll<w; lll++){
              if (field[index+lll]>partition)
              balance++;
            }

            If median is found, break;
          }                          [4]
      }
  index = (partition + hops) % (f-w)
  hops++;
  }
}
```

Figure 1: Kernel of Pointer Stressmark

$P(0.5(w(w+1)))$ memory accesses will be generated, where P is a factor less than 1. P comes out because of the conditionals (2)and (3). If (2) is satisfied, no more memory accesses will be performed in this hop. Otherwise, if (3) is satisfied, the $(w-l)$ elements following the element l will be scanned, where l is the position of the current element in the window. In the worst case, all the elements in the window would be scanned. So, at most $0.5(w(w+1))$ memory accesses will be generated in each hop. However, because of the probability that (2) can be true, a smaller number of memory accesses will be generated, that is $P(0.5(w(w+1)))$. Finally, we get:

$$\text{Number of Memory accesses} = \#thread \bullet \#hops \bullet (P(0.5(w(w+1)))$$

In our estimation, P is chosen to be 0.5. So $0.25(w(w+1))$ of memory accesses will be generated in each hop.

3.1.2 L1 Data Cache Misses and Optimization

Since the elements in a window with size of w is scanned sequentially, the spatial locality is exploited. Also,the temporal locality inside the window is well exploited because the window is scanned sequentially. However, the spatial locality inside the window is not fully exploited because of the compulsory misses and capacity misses (conflict misses are caused by capacity misses). Since L1 line size is 32 Bytes and each element is 4 Bytes (unsigned int), for a window with size w=15, at least 2 and at most 3 misses are generated in a window (if the referred value is at the boundary of the cache blocks, only 2 misses will be generated). As a result:

Number of L1 cache Misses = $\#thread \bullet \#hops \bullet (2 \sim 3)$

Optimization: One way to reduce L1 data cache misses is $prefetch$, by which both compulsory misses and capacity misses will be reduced. Either SW or HW prefetch can be employed. In the case of SW prefetch, the next 2 blocks are prefetched when the first block of the sampled window is accessed. Up to 2 misses will be reduced for each window. If the reference stream of a window covers 3 blocks, 60% cache misses will be reduced. In the case of HW prefetch, a stream buffer with two 32-byte entries is enough. Also, 60% reduction of cache misses is expected.

Victim cache will not help much here because the portion of conflict misses is very small in the total L1 data cache misses.

3.1.3 L2 Data Cache Misses and Optimization

Because of the large size of L2 cache,capacity misses and conflict misses are reduced relative to those of the L1 data cache. This reduction of capacity misses is significant when the data field size is small. Intuitively, the smaller the size of field, the smaller the number of capacity misses and conflict misses will be, and the higher the probability that the data will be reused before being replaced. For inputs 3, 4, 6, which have relative small field size of 1648577, the miss rate reduction is more significant than that of the other inputs with larger field size. 1 or 2 misses would be generated each time a median is searched for, because the block size is 64B now. Finally, we can estimate the L2 cache misses by:

Number of L2 Cache misses = $\#thread \bullet \#hops \bullet (1 \sim 2)$

Please note that the above equation is accurate only when the field size is very large. When the field size is small, it is not accurate because a lot more conflict misses are generated. the number of which is difficult to estimate.

Optimization: Either SW or HW prefetch is effective in reducing compulsory misses and capacity misses. For SW prefetch, about 50% of the L2 cache misses can be eliminated if the next block is prefetched when the first block of the window is accessed. In the case of HW prefetch, stream buffer with at least one 32B entry can be employed. The performance improvement can be up to 50%.

3.2 Update Stressmark

Update stressmark was not analyzed here because its memory access pattern is almost the same as that of Pointer Stressmark. The only difference between Update Stressmark and Pointer Stressmark is that there is a updating operation after each median is found. However, the similar simulation statistics shows that this updating operation will not affect the memory behavior notably.

3.3 Matrix Stressmark

3.3.1 Stressmark Kernel and Source of Memory Accesses

Matrix Stressmark requires solving the equation $Ax = b$ for vector x, where A is matrix, x and b are vectors. BiConjugateGradient Algorithm is utilized to approximate the solution. The algorithm is iterated until x is found to be within a specified error tolerance or a fixed number of iterations have been done. Compact Row Storage scheme is used to store the sparse matrix, which greatly reduces the computation time and memory traffic.

```
while((iterations < maxIterations)
      && (error > errorTolerance))
{
 matrixMulVector();

vectorSub();
vectorAdd();      Vector
vectorMul()       Operations
...

matrixMulvector();

iterations++;
}
```

(a) Loop body

```
void matrixMulVector(double *value,
                     int *col_ind,
                     int *row_start,
                     double *vector,
                     double *out)
{
   ...
   for (l=0;l<dim; l++){
      *(out + l) = 0;
      tmp_rs = row_start[l];
      tmp_re = row_start[l+1];
      for (ll = tmp_rs; ll<tmp_re; ll++){
        *(out + l) += value[ll] * vector[ col_ind[ll] ];
      }
   }
}
```

(b) Matrix * Vector

```
void vectorSub(double *vector1,
               double *vector2,
               double *vector){
  for (l=0; l<dim; l++){
    *(vector + l) = *(vector1+l) - *(vector2+l);
  }
}
```

(c) Vector Operations

Figure 2: Kernel of Matrix Stressmark

The kernel of Matrix Stressmark includes two kinds of operations, as shown in Figure 2: one is the multiplication of Matrix and vector: $matrixMulvector$, the other is the operation between vectors, such as vector addition, vector multiplication, and so on. In the kernel loop body, $MatrixMulvector$ is called twice and a set of vector operations are invoked between two $MatrixMulvector$s. Up to $\#maxIteration$ iterations of the loop body can be executed in each run.

Memory accesses are generated when accessing matrix and vectors in $matrixMulVector$ and vector operations. By adding the corresponding portions of the memory accesses in $matrixMulvector$ and vector operations, we get:

$$\text{Number of Memory Accesses generated} = \underbrace{2(6dim + 5NumberNonzero)}_{FromMatrixMulVector} + \underbrace{42dim}_{Fromvectoroperations}$$

$NumberNonzero$ is generally to be much larger (10 times or more) than dim, so that most of the memory accesses are generated by accessing matrices.

3.3.2 L1 Data Cache Misses and Optimization

The spatial locality is high in Matrix Stressmark because the reference stream of the matrices and most of the vectors is sequential. Compulsory misses and capacity misses are the major source of L1 cache misses because of the small size of L1 data cache. Conflict misses exist between vectors but they can ignored because the reference stream for vectors is seldom interleaved with each other. Since each element in matrix and data vectors is 8 Bytes (typed double), every 4 cache accesses will generate a cache miss. Considering all the factors, we get:

$$\text{Number of L1 Cache Misses} = \underbrace{2(0.75dim + 0.375N + P(N + dim))}_{FromMatrixMulVector} + \underbrace{11.5dim}_{Fromvectoroperations}$$

Here N is $NumberNonzero$, P is a factor less than 1. P is used to describe the indeterminism of the accesses to $vector[col\_ind[ll]]$. Even though col_ind is accessed sequentially, the trace of accessing $vector$ may jump back and forward, depending on the value of $col\_ind[ll]$. Some conflict misses are generated because of this. We chose a factor P to describe this situation.

It can be seen from the above equation that most of the cache misses are generated by calling $matrixMulVector$. It is accesses to matrix that generates most of the L1 data cache misses. So how to reduce the misses generated by accessing matrix is the key point.

Optimization: Prefetch is effective in reducing compulsory misses and capacity misses since no matter whether matrix or vectors is accessed, the trace is always sequential.

Both SW and HW prefetch are effective in cache optimization. $Stream\_Buffer$ is employed as HW prefetch. $Stream\_Buffer$ tries to eliminate all the compulsory misses and capacity misses. Its effectiveness is shown in the simulation result.

No efficient way can deal with the portion represented by $P(N + dim)$ because of its indeterminism.

3.3.3 L2 Data Cache Misses and Optimization

Capacity misses are greatly reduced because of large L2 cache size. Also, conflict misses among vectors are reduced because vectors can now co-exist in the cache. However, different input sizes (determined by the input parameters Dimension and NumberNonzero) may result in different memory behavior.

When the input size is relative small, such as inputs 1 and 2, miss rates are small because most of the cache misses are compulsory misses. Capacity misses and conflict misses are insignificant in this case because the total size of the Matrix and vectors ($60dim + 12NumberNonzero$) is less than 1MB.

When the input size grows large gradually, such as inputs 3 and 4, the portion of capacity misses gradually increases because the size ($60dim + 12NumberNonzero$) exceeds 1MB. Most of these misses happen on matrix because of its large size. The spatial/temporal locality still exist for vectors because the total size of vectors is still small.

When the input size becomes far larger than 1MB, such as inputs 5, 6, and 7, the portion of capacity misses becomes dominant. These capacity misses are mainly generated by accessing matrix. The spatial/temporal locality still exits for vectors. The indeterminism of $P(N + dim)$ is now eliminated because all the vectors can completely be accommodated. Finally, in this case:

Number of L2 Cache misses (large input)= $2((3/8)dim + (3/16)NumberNonzero)Iterations$

Optimization: Prefetch is effective in reducing compulsory misses and capacity misses for the same reason applied on L1 cache. We will show its effectiveness in our simulation.

3.4 Transitive_Closure Stressmark

3.4.1 Stressmark Kernel and Source of Memory Accesses

The goal of Transitive_Closure stressmark is to find out the solution of the *All_Pairs_Shortest_Path* problem.The kernel is show in Figure 3. The time complexity of the algorithm is $O(N^3)$ because

```
        ...
for (k=0; k<n; k++){
      ...
   for (i=0; i<n; i++){
      for (j=0; j<n; j++){
         old = *(din + j*n + i);
         new1 = *(din + j*n + k) + *(din + k*n + i);
         *(dout + j*n + i) = (new1 < old? new1: old);              (1)
          assert (*(dout + j*n + i) <=NO_PATH);
          assert (*(dout + j*n + i) <= *(din + j*n + i));
      }
   }
   dtemp = dout;
   dout = din;
   din = dtemp;
}
```

Figure 3: Kernel of Transitive_Closure Stressmark

the loop body (1) is iterated N^3 times. In each iteration, at least 3 values have to be accessed: $*(din + j * n + i), *(din + j * n + k), and * (dout + j * n + i)$. The loop invariant $*(din + k * n + i)$ will be in register when needed. It can be estimated that the number of valid memory accesses is:

Number of Valid Memory Accesses = $3N^3$

Actually, more memory accesses are generated for address calculation. For example, to get the address (din + j*n + i), the base address din, j, and i have to be loaded from memory. Besides this, the assertion tests generates more memory accesses, finally, we get:

Number of memory access = $19N^3$

Please note that different compilers may generate different number of memory accesses, but the valid number of memory accesses should always be $3N^3$.

3.4.2 L1 Data Cache Misses and Optimization

Almost all the L1 cache misses are compulsory misses and capacity misses. The spatial locality in the cache block is destroyed because of the following reason. In the loop body, each time when j is increased by 1, the memory addresses $(din + j * n + i), (din + j * n + k)$, and $(dout + j * n + i)$ are increased by N. Only one element in the corresponding blocks is referenced in each iteration. After that, the reference stream moves forward by N. These blocks will not be referenced again until j starts from zero again. By that time, however, the useful blocks will have long before being replaced. For example, suppose these blocks don't conflict with each other, after about $(512 \bullet 2)/3 = 341$ iterations, the old blocks will start being replaced. After 341 * 2 = 682 iterations, no old blocks will exist. Next time when the innermost loop is executed, all these required blocks would have to be re-fetched again. So accesses to $*(din + jn + i), *(din + jn + k) and * (dout + jn + i)$are always missed. Thus:

Number of L1 Cache Misses = $3 \bullet N^3$

Optimization: Compiler techniques is preferred to architectural ones in case of Transitive_Closure stressmark. Compiler can do much better job and more efficiently than architectural techniques.

One of the compiler techniques is $Blocking$ $(Tiling)$. In this technique, the big working set is divided into smaller sets (blocks). These small blocks can be accommodated by the small data cache individually although the original large working set is too large to be held. Not only are the capacity misses reduced, but also the temporal locality in each block can be fully exploited. The Figure 4 gives a simple blocking example, in which H represents the height of the block (We choose N=1100 as input). Our simulation (after removing unnecessary assertions) shows that when $H = 100$, the miss rates reduction is 78%; when $H = 50$, the reduction is 87%. More efficient blocking ways will be further explored.

```
          ...
for (k=0; k<n; k++){
        ...
   for (p=0; p<n; p+=H){
     for (i=0; i<n; i++){
        for (j=p; j<((p+H)<n? (p+H):n); j++){
            old = *(din + j*n + i);
            new1 = *(din + j*n + k) + *(din + k*n + i);
          *(dout + j*n + i) = (new1 < old? new1: old);
     }
   }

 }
 dtemp = dout;
 dout = din;
 din = dtemp;
}
```

Figure 4: Blocking Applied on Transitive_Closure Stressmark

3.4.3 L2 Data Cache Misses and Optimization

Since the size of L2 data cache is very large, most of the misses should be compulsory misses. Without taking conflict misses into account, the number of capacity misses is very small which can be estimated as follows. In each inner loop, the trace will cover about at most $3N$ cache blocks. Since all the inputs have N less than 2000, at most 6,000 blocks are covered, which is less than the number of cache blocks that can be accommodated by L1 data cache: 1MB/64B = 16384 (without taking conflict misses into account). This means that almost all these blocks can be reused before being replaced in a loop. Since each L2 cache block is of size 64B, and each data is 4B (unsigned int), 1 miss occurs every 16 iterations. So if without conflict misses, we have:

Number of L2 Cache Misses = $2 \bullet (N^3/16)$

Unfortunately, conflict misses exist because of mapping problem. For example, for input 6, 7, 8 with $N = 1250$, for every $16N$ loops, $800 \bullet 15$ blocks will be missed because of conflict. Considering this, the number of misses is estimated to be $2 \bullet ((N + 800 \bullet 15)N^2/16)$, which increases the miss rates.

Optimization: The effectiveness of blocking is not notable in L2 cache optimization because most of the cache misses are now compulsory misses, the ratio of which is already very small.

Stream buffer is used for L2 cache optimization. The next block which is prefetched is the block at the distance of $N \bullet 64$. However, the commonly used one-way stream buffer does not work here, and almost no misses can be removed. The reason is that data references tend to consist of interleaved streams of data from different matrices or different parts of a matrix. Thus, a multi-way stream buffer was employed. From the simulation, it is found that when a stream buffer with more than 2 ways is employed, 52.5% of misses can be removed.

3.5 Field Stressmark

3.5.1 Stressmark Kernel and Source of Memory Accesses

The Field Stressmark involves searching a field with size of f for token strings. If a matching instance is found, some information will be updated and elements in the instance is modified. The next searching will start at a distance of *token_length*. If no instance is found, some simple state information is updated, and the sampling window moves forwards by 1. For every token, this process is performed until the whole field is scanned. Then the same process starts for the next token. The kernel of Field Stressmark is shown in Figure 5.

We assume that the probability of finding a matching instance is very small. That is, we assume (2) is rarely satisfied. Since the value of each item in the token can be any one of the 256 values, and the token length is set to be 7, the probability of finding a matching instance is: P = $(1/256)^7$, This is a very small value. This in turn means that (3) is frequently satisfied.

In the process of testing for a matching instance, which is shown in (1), items in the data

```
for (l=0; l<n; l++){
   token[l].subfields = 0;
   token[l].stat[0].count = 0;
   token[l].stat[0].sum = 0;
   token[l].stat[0].min = MAX_TOKEN_VALUE;
   index = 0;
   while ((index<f) && (token[l].subfields < MAX_SUBFIELDS)){(1)
        offset = 0;
        while ((field[index+offset]==token[l].delimiter[offset])
                && (offset < token[l].length)){
            offset++;
         }
  if (offset == token[l].length){            (2)
        for (offset=0; ofset<token[l].length;  offset++){
               field[index+offset} = field[index+offset} + field{index+offset+mod_offset)%f}
                                     %(MAX_TOKEN_VALUE);
         }
        index += token[l].length-1;
        token[l].subfields++;
 else {                                  (3)
        token[l].stat[token[l].subfields].count++;
        token[l].stat[token[l].subfields].sum += field[index];
        if(token[l].stat[token[l].subfields].min > field[index])
        token[l].stat[token[l].subfields].min = field[index];
 }
 index++;
 }
}
```

Figure 5: Kernel of Field Stressmark

field and token must be compared with each other until an unmatched item is encountered. Generally, less than the $token\_length$ of items are scanned before failed. So we designate the average scanning length is $P \bullet Token\_length$, where P is less than 1. So we get:

Number of Memory accesses = $\#token \bullet Sizeoffield * [P \bullet Token\_length \bullet (\#mem\_involved) + P(match) \bullet (\#mem\_involved) + P(unmatch) \bullet (\#mem\_involved)]$

In our estimation, we choose P = 0.5, P(match)=0.01 and p(unmatch)=0.99.

3.5.2 L1 Data Cache Misses and Optimization

Most of the misses in L1 data cache misses are compulsory misses and capacity misses since the reference stream is almost sequential and L1 cache size is small. The number of conflict misses is relative small because the trace is seldom intertwined. Since (2) is rarely satisfied, $token[l].subfield$ is seldom changed. So the same record $token[l].stat[token[l].subfields]$ are reused frequently when (3) is satisfied, which means small number of misses happen on it. Also, the temporal locality in each cache block is high because every element in it is repeatedly accessed.

Since each element is 1B (unsigned Char), 1 miss occurs on every 32 different element accesses. Thus:

Number of L1 Cache Misses = $\#token \bullet Sizeoffield * 1/32$

Optimization: The already good L1 and L2 cache performance can be further improved by prefetching. As we have explained, $Stream\_Buffer$ is is an efficient way to reduce compulsory and capacity misses. The performance improvement can be up to 100% if not taking conflict misses into account.

3.5.3 L2 Cache Misses and Optimization

The analysis is almost the same as that of L1 cache. Since the cache block is 64B, we get:

Number of L1 Cache Misses = $\#token \bullet Sizeoffield * 1/64$

Still, $Stream\_Buffer$ is employed for optimization. For the same reason as that for L1 data cache, theideal performance improvement can be up to 100%.

3.6 Neighborhood Stressmark

3.6.1 Stressmark Kernel and Source of Memory Access

Neighborhood stressmark estimates the texture of a gray-level co-occurrence matrix (GLCM). It estimates the scale of the texture by measuring the GLCM energy and GLCM entropy for each of the four directions at two distances. The distances are given as the input parameters $distanceShort$ and $distanceLong$. The directions are constant for all tests: 0^0(horizontal),45^0(right diagonal), 90^0(Vertical), and 135^0(left Diagonal). In each test, up to $(rowHigh - rowLow) \bullet (colHigh - colLow)$ pixels are scanned, where rowHigh, rowLow, colHigh, and colLow are decided by $dimension$ and $distanceShort$ or $distanceLong$.

```
                                                                          Memory access complexity
  ...
for (index = 0; index < numBins; index++){
     sumHist[index] = 0;                                         (1)           O(numBins)
     diffHist[index] = 0;
}
   ...
value0RowOffset = rowLow * dimension;
value1RowOffset = (rowLow + dy) * dimension;
for (rowIndex = rowLow; rowIndex<rowHigh; rowIndex++){
    for (colIndex = colLow; colIndex<colHigh; colIndex++){
  ...
            colForPixelAtDistance = colIndex + dx;
            value0 = *(image + value0RowOffset + colIndex);
            value1 = *(image + value1RowOffset + colForPixelAtDustance);
            binIndex = value0 + value1 - 2*MIN_PIXEL;
            assert((binIndex >= 0) && (binIndex < numBins));           O((rowHigh-rowLow))
            sumHist[binIndex] += 1;                              (2)
            binIndex = value0 - value1 + maxPixel -MIN_PIXEL;
            assert((binIndex >= 0) && (binIndex < numBins));
            diffHist[binIndex] += 1;

            totalNumPixels += 1;
    }
   value0RowOffset += dimension;
    value1RowOffset += dimension;
}
     ...
for (index=0; index<numBins; index++){
         ....                                                    (3)           O(numBins)
 }
```

Figure 6: Kernel of Neighborhood Stressmark

The kernel of Neighborhood Stressmark is shown in Figure 6. The kernel is divided into 3 parts with different memory access complexity: $O(numBins)$ for (1) and (3), $O((rowHigh - rowLow)(colHigh - colLow))$ for (2). In the program based on our inputs, $numBins$ is much smaller than $(rowHigh - rowLow)(colHigh - colLow))$, generally more than 1000 times. So the memory behavior of the parts (1) and (3) can be ignored. We can focus on the memory behavior of part (2).

There are 6 valid memory accesses in (2): 2 for $value0$ and $value1$, 2 for $sumHist[binIndex]+ = 1$, and 2 for $diffHist[binIndex]+ = 1$. Finally, we can estimate:

Number of valid Memory Accesses =
$\sum_{distShort,distLong} \sum_{0^0,45^0,90^0,135^0} 6(RowHigh_i - rowLow_i)(colHigh_i - colLow_i)$

Actually, some more memory accessed are generated for loading base address of arrays and address calculation.Finally, $13 \sim 14$memory accesses are generated for (2).

3.6.2 L1 Data Cache Misses and Optimization

The misses generated by parts (1) and (2) can be ignored because of the relative small value of $numBins$. In part (3), the misses generated by $sumHist[binIndex]+ = 1$, and $diffHist[binIndex]+ = 1$ can also be ignored. The temporal locality of these two arrays is high because of the small size $numBins$. The spatial locality is also high because the reference stream is always to "neighbors". We can figure out that the variation of indexes for $sumHist$ and $diffHist$ is generally small because these indexes are the function of $(value0, value1)$. The $value0$ and $value1$ are the values of image pixels, which change smoothly normally. So the spatial locality of $sumHist$ and $diffHist$ is high.

L1 cache misses are mainly generated by accessing image matrix. These are mainly compulsory misses and capacity misses. There is little chance for $value0$ and $value1$ to reuse themselves individually because of large space separation. Also, the probability for $value0$ and $value1$ to reuse each other is also very small in the cases of $45^0, 90^0, 135^0$. The large value of $dimension$ multiplied by $rowOffset$ would make either of them evicted. In the case of 0^0, however, the temporal locality is somewhat exploited because the row offset is zero now such that the address difference between these two is at most $dimension$, which is small enough for either of them being reused. Since each element is of 2B (typed short), 1 miss is generated every 16 memory accesses. Taking all these into account, we get:

Number of L1 Cache Misses = $\sum_{45^0,90^0,135^0} 2(RowHigh_i - rowLow_i)(colHigh_i - colLow_i)/16 + \sum_{0^0}(RowHigh_i - rowLow_i)(colHigh_i - colLow_i)/16$

Optimization: Prefetch is still effective in reducing compulsory misses and capacity misses. $Stream\_Buffer$ is still preferred because of its efficiency. 100% reduction of cache misses is expected if we ignore the effect of conflict misses.

As a fine tune, $sumHist$ and $diffHist$ can be pinned in the cache so that its temporal locality is kept. Or another way is to use $Victim\_Cache$ to hold their victims.

3.6.3 Number of L2 cache misses

The same analysis can be employed for L2 cache. Now, the misses generated by accessing sumHist and diffHist can totally be ignored because the large L2 cache can exploit more of their temporal/spatial locality. Now the block size is 64 B, so every 32 memory As a result:

Number of L1 Cache misses = $\sum_{45^0,90^0,135^0} 2(RowHigh_i - rowLow_i)(colHigh_i - colLow_i)/32 + \sum_{0^0}(RowHigh_i - rowLow_i)(colHigh_i - colLow_i)/32$

Optimization: $Stream\_Buffer$ is enough in reducing L2 cache misses to a very small ratio. Still, 100% reduction of cache misses is expected. It is unnecessary to deal with conflict misses now because they can be totally ignored.

4 Conclusion

In this report, each of the 6 DIS Stressmarks is analyzed in detail concerning the memory behaviors and L1/L2 data cache misses. Possible cache optimization techniques were discussed.

We can see that $prefetch$ technique is effective in programs dominated by matrix and vector computations as well as with purely sequential reference streams. In matrix and vector computations, the size of these data structures is so large that capacity misses are dominant. Conflict misses are insignificant in DIS stressmark suite.

Sparse Matrix computation further complicates our analysis in term of its storage scheme and the distribution of Nonzero elements. Compact Row Storage scheme was employed in our implementation of BiConjugateGradient algorithm. Although computation complexity and memory bus traffic are compressed, indeterminism is the negative side effect in this scheme. We are doing research for the efficient solutions of this problem.

Transitive_Closure Stressmark has large miss rates because of the capacity misses. In the case of L1 data cache in which the working set is relative large, *blocking* is effective in reducing the capacity misses and exploiting the temporal locality in each block. For L2 Cache, however, prefetching is more efficient than Blocking since most of the misses are compulsory misses.

We are now doing simulation to verify the effectiveness of the optimization methods proposed in this report.

References

[1] Norman P. Jouppi,*Improving Direct-Mapped Cache Performance by the Addition of A Small Fully-Associative Cache Prefetch Buffers.* International symposia on Computer architecture (selected papers),1998, Pages 388 - 397.

[2] Chiou, D.; Jain, P.; Rudolph, L.; Devadas, *Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches.* Proceedings 2000. Design Automation Conference.Proceedings of ACM/IEEE-CAS/EDAC Design Automation Conference, Los Angeles, CA, USA, 5-9 June 2000.) New York, NY, USA: ACM, 2000. p.416-19..

[3] Panda, P.R.,Nakamura, H., Dutt, N.D., Nicolau,A. *Augmenting Loop Tiling With Data Alignment For Improved Cache Performance.* IEEE Transactions on Computers, vol.48, (no.2), IEEE, Feb. 1999. p.142-9.