Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1575–1582

# Dynamic loop reversal - the new code transformation technique

I. Šimeček, P. Tvrdík
Department of Computer Systems, Faculty of Information Technology,
Czech Technical University in Prague Prague, Czech Republic Email: xsimecek,pavel.tvrdik@fit.cvut.cz

***Abstract*—In this paper, we describe a new source code transformation called *dynamic loop reversal* that can increase temporal and spatial locality. We also describe a formal method for predicting the cache behaviour and evaluation results of the accuracy of the model by measurements on a cache monitor. The comparisons of the numbers of measured cache misses and the numbers of cache misses estimated by the model indicate that model is relatively accurate and can be used in practice.**

## I. Introduction

LINEAR codes for dense linear algebra consist mainly of loops. A number of source code transformations techniques have been developed and used in the state-of-the-art compilers. In this paper, we consider the following standard techniques: *loop unrolling*, *loop blocking*, *loop fusion*, and *loop reversal* [1], [2], [3]. The main result of this paper is a description of a new transformation technique, called *dynamic loop reversal*, shortly *DLR*, to improve temporal and spatial locality.

Models for predicting the number of cache misses have also been developed for standard source code transformations [4], [5], [6], [7]. In order to incorporate the DLR into compilers, we propose such a model for the DLR in Section IV-B .

## II. Terminology

Throughout the paper, we assume that indexes of vectors and matrices start from 1, all elements of vectors and matrices are of type *double* and that all matrices are stored in the row-major format.

### A. The cache architecture model

We consider a *set-associative cache*. The number of sets is denoted by $h$. One set consists of $s$ independent *blocks*. The size of the data part of a cache in bytes is denoted by $DC_{\mathrm{S}}$. The cache block size in bytes is denoted by $B_{\mathrm{S}}$. Then $DC_{\mathrm{S}} = s \cdot B_{\mathrm{S}} \cdot h$. The size of type `double` is denoted by $S_{\mathrm{D}}$. We consider only *write-back* caches with *LRU block replacement* strategy.

### B. The compressed sparse row (CSR) format

A matrix $A$ is *dense* if it contains $\Theta(n^2)$ nonzero elements and it is *sparse* otherwise. In practice, a matrix is considered sparse if the ratio of nonzero elements drops bellow some threshold. The most common format (see [8], [9], [10]) for storing sparse matrices is the *compressed sparse row* (CSR) format. The number of nonzero elements is denoted by $NZ_A$. A matrix $A$ stored in the CSR format is represented by three linear arrays $Elem_A$, $Addr_A$, and $Ci_A$. Array $Elem_A[1,\ldots,NZ_A]$ stores the nonzero elements of $A$, array $Addr_A[1,\ldots,n]$ contains indexes of initial nonzero elements of rows of $A$, and array $Ci_A[1,\ldots,NZ_A]$ contains column indexes of nonzero elements of $A$. Hence, the first nonzero element of row $j$ is stored at index $Addr_A[j]$ in array $Elem_A$. The density of the matrix $A$ (denoted by $density(A)$) is the ratio between $NZ_A$ and $n^2$.

## III. Code restructuring

In this section, we propose a new optimization technique called *dynamic loop reversal* (or alternatively *outer-loop-controlled loop reversal*).

### A. Standard static loop reversal

In the standard loop reversal, the sense of the passage through the interval of a loop iteration variable is reversed. This rearrangement changes the sequence of memory requirements and reverses data dependencies. Therefore, it allows further loop optimizations in general.

**Example code 1**

```
1: for i ← n, 2 do
2:     B[i]+ = B[i − 1];
3: for i ← 2, n do
4:     A[i]+ = B[i];
```

Example code 1 represents a typical combination of data-dependent loops whose data dependency can be recognized automatically by common compiler optimization techniques. However, the first loop is *reversible* (it means that it is possible to alternate the sense of the passage). The reversal of the second loop and loop fusion can be applied and the reuse distances (see Section IV-A for the definition of the reuse distance) for memory transactions on array $B$ are decreased.

**Example code 2** Loop reversal and loop fusion applied to Example code 1

```
1: for i ← n, 2 do
2:     B[i]+ = B[i − 1];
3:     A[i]+ = B[i];
```

In Example code 3, data-dependency analysis reveals that the two loops are also reversible.

978-1-4673-4471-5/$25.00 

**Example code 3**

```
1: for i ← 1, n do                    ▷ Loop 1
2:     s+ = A[i] ∗ A[i];
3: norm = √s;
4: for i ← 1, n do                    ▷ Loop 2
5:     A[i]/ = norm;
```

However, the application of the loop reversal to the second loop decreases the reuse distances.

**Example code 4** Loop reversal applied to Example code 3

```
1: for i ← 1, n do                    ▷ Loop 1
2:     s+ = A[i] ∗ A[i];
3: norm = √s;
4: for i ← n, 1 do                    ▷ Loop 2
5:     A[i]/ = norm;
```

The problem is that in this case (and in other similar cases), the compiler heuristics for the decision which loop to reverse to minimize reuse distances is complicated.

### B. The effect of the static loop reversal on cache behaviour

If the size of array $A$ is less than the cache size ($nS_{\mathrm{D}} \leq DC_{\mathrm{S}}$), then Example codes 3 and 4 are equivalent as to the cache utilization. However, if the size of array $A$ exceeds the cache size, then no elements of $A$ are reused in Example code 3,whereas the last $k = \frac{B_{\mathrm{S}}}{S_{\mathrm{D}}}$ elements of $A$ are reused within the second reversed loop in Example code 4. So, the loop reversal improves the temporal locality (see Figures 1 and 2).

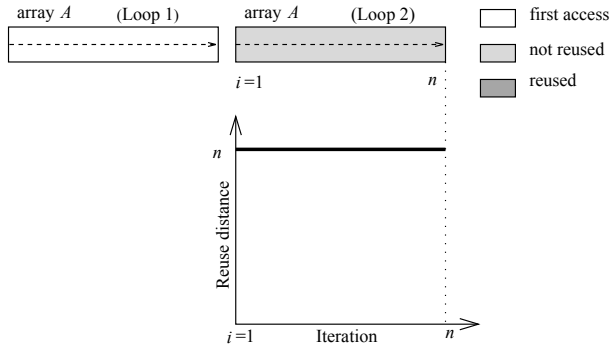


Fig. 1. The reuse distances in Example code 3.

### C. Dynamic loop reversal

The static loop reversal is used to reverse data-dependency in one dimension. This has motivated us to generalize this idea and we have designed another optimization for nested reversible loops based on loop reversal. Consider the following code:

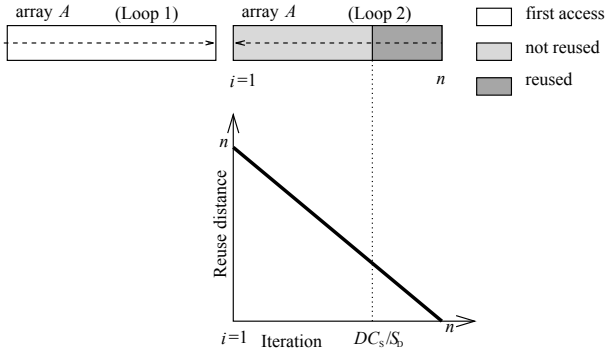


Fig. 2. The reuse distances in Example code 4.

**Example code 5**

```
1: for i ← 1, n do
2:     s = 0;
3:     for j ← 1, n do
4:         s+ = A[i][j] ∗ x[j];
```

The direction of the inner loop can be alternated forward and backward in even and odd iterations of the closest outer loop. In this way, we can use the positive effect of a loop reversal in every iteration of the outer loop. This is why we call it a *dynamic loop reversal*, or *DLR* for short. Example code 5 is a candidate for such a transformation.

**Example code 6** DLR applied to Example code 5

```
1: for i ← 1, n do
2:     s = 0;
3:     if i is odd then
4:         for j ← 1, n do
5:             s+ = A[i][j] ∗ x[j];
6:     else
7:         for j ← n, 1 do
8:             s+ = A[i][j] ∗ x[j];
```

We will denote this transformation by DLR($i \rightarrow j$). For large arrays, this transformation leads to even better temporal locality than the original Example code 5, because it reduces the reuse distances and data reside in the cache from the previous iteration. On the other hand, the saving of the number of cache misses in one iteration of the outer loop is bounded by $DC_{\mathrm{S}}/B_{\mathrm{S}}$. So, the *DLR* has a significant effect if the cache size is comparable to the sum of affected arrays sizes in one iteration of the outer loop. The necessary condition for applying the *DLR* is that the inner loop must be reversible.

### D. The application of DLR on triple-nested loops

In the previous text, the DLR was applied to double-nested loops, but it can also be applied to triple-nested loops. Consider the following code skeleton:

**Example code 7**

```
1: for i ← 1, n do
2:     for j ← 1, n do
3:         for k ← 1, n do
4:             (* loop body *)
```

In Example code 7, there are three options how the DLR can be applied:

- on the $i$-loop: DLR($i \to j$),
- on the $j$-loop: DLR($j \to k$),
- both transformations: DLR($i \to j$) and DLR($j \to k$).

The last option means composition of two transformations DLR($i \to j$) and DLR($j \to k$). This composition we will denoted by DLR($i \to j \to k$). In this case, the effect of DLR is twofold: DLR($i \to j$) (on the outer pair of loops) can improve temporal locality inside the L2 cache and DLR($j \to k$) (on the inner pair of loops) can improve temporal locality inside the L1 cache.

### *E. Comparison and possible combinations of DLR and other loop restructuring techniques*

In this section, we describe some loop restructuring techniques (for details see [11], [12], [13], [14], [15]), compare them with DLR, and discuss their possible combinations with DLR.

*1) Loop unrolling:* Loop unrolling has two main effects. Firstly, it makes the sequential code longer, so it may improve data throughput, because the instructions could be better scheduled and the internal pipeline could be better utilized. Secondly, the number of test condition evaluations drops according to the unrolling factor. In general, the loop unrolling concentrates on maximizing the machine throughput, not on improving the cache behaviour.

*2) Loop tiling (blocking):* Loop tiling (sometimes called loop blocking or iteration space tiling) is one of advanced loop restructuring techniques. A compiler can use it to increase the cache hit rate. One possible motivation for using this technique is that the *loop range* (e.g., the size of the array traversed repeatedly within the loop) is too big and exceeds the data cache size $DC_{\mathrm{S}}$. Thus, the loop should be split into two loops: the outer loop is the out-of-cache loop and the inner one is the in-cache loop. The value $B_f$ is called the *tiling or block factor* and its optimal value depends on the size of the cache.

The loop tiling and DLR can be easily combined. DLR can be applied on every pair of immediately nested loop, but its useless to apply it for in-cache loops ($i$-loop, $j$-loop, and $k$-loop). We consider loop tiling as a competitor for DLR and we have performed experiments with both. These quantitative measurements of effects of these techniques are presented in Section VI-D.

## IV. An Analytical Model of the Cache Behavior for the DLR

The *polytope model* (for details see [3], [6]) is used by modern compilers for an estimation of the parameters for loop restructuring techniques. We will present two cache behaviour models based on reuse distances (shortly RD).

### *A. A cache miss model with reuse distances*

This model is inspired by the model introduced in [16]. We will call it the *basic RD model*.

**Definition** Consider an execution of an algorithm on the computer with load/store architecture and assume that addresses of memory transactions during this execution form a sequence $P[1,\ldots,n] = [addr_1,\ldots,addr_n]$. Then $P$ is called a **sequence of memory access addresses** and $P[i] = addr_i$ is the $i$-th transaction with memory address $addr_i$. The **reuse distance** $RD(t)$, where $t \in \langle 1, n\rangle$, is the number of **different** memory addresses accessed between two uses of the address $P[t]$. Formally, if $P[t] = addr_t$ and $\epsilon(t) > 0$ is the minimal integer number such that $P[t-\epsilon(t)] = addr_t$, then $RD(t) = |\{P[t-\epsilon(t)],\ldots,P[t-1]\}|$. If such an $\epsilon(t)$ does not exist, then $RD(t) = \infty$, otherwise $RD(t) \le \epsilon(t)$.

The notion of reuse distances can be used for developing a simple cache miss model based on estimating the numbers of thrashing misses in fully-associative ($h = 1$) caches. If $RD(t) > DC_{\mathrm{S}}/S_{\mathrm{D}}$, then the content of the cache block from the memory address $P(t)$ is replaced by some new value and a cache miss occurs. If $RD(t) = \infty$, then a compulsory miss occurs, otherwise a thrashing miss occurs. Recall that we assume only caches with LRU block replacement strategy.

In this basic RD model, the spatial locality of the cache memory is not considered, i.e., it is assumed that a cache block contains exactly one array element ($B_{\mathrm{S}} = S_{\mathrm{D}}$). However, $B_{\mathrm{S}} = c \cdot S_{\mathrm{D}}$, where $c$ is typically 4 or 8 in modern processors, and therefore, spatial locality must be taken into account in order to have a more realistic model.

### *B. A simplified cache miss model for the DLR*

Even the basic RD model is too complicated for modelling the cache behavior of DLR in real applications. Hence, we introduce another model that is even more simplified. We call this model *simplified RD model*. We use this model for enumeration cache misses saved by DLR. To derive an analytical model of the effect of the DLR on the cache behaviour, consider the following code skeleton representing most often memory access patterns during a matrix computation:

**Example code 8**

```
1: statement_1;
2: for i ← i_1, i_2 do
3:     statement_2;
4:     for j ← j_1, j_2 do
5:         statement_3;
6:         = B[j];          ▷ Memory operation of type α
7:         = B[i];          ▷ Memory operation of type β
8:         = A[i][j];       ▷ Memory operation of type γ
9:         = A[j][i];       ▷ Memory operation of type δ
10:    statement_4;
11: statement_5;
```

We consider the following simplifying conditions:

A1 We assume that all matrices are stored in the row-major order.

A2 We assume that *statements*$_{1-5}$ contain only local computation with register operands. That is, we assume that *statements*$_{1-5}$ have negligible cache effects and the only memory accesses are memory operations of type $\alpha - \delta$.

A3 We assume that the reuse distances depend on the exact ordering of memory operations (inside the $j$-loop) only slightly and so do the number of cache misses.

A4 We do not distinguish between load and store operations.

A5 We assume that the cache memory is big enough to hold all the data for one iteration of the (inner) $j$-loop.

A6 We assume that the cache memory is not able to hold all the data for one iteration of the outer $i$-loop. Otherwise, the DLR has no effect in comparison to standard execution.

A7 This model is derived only for immediately nested loops.

Let us now analyse the effect of DLR($i \to j$) on individual memory operations.

- A memory operation of type $\alpha$ is affected by the DLR, because its operand (or its part) can be reused. The effect of DLR can be estimated by the RD analysis.
- A memory operation of type $\beta$ is not affected by the DLR, because it returns the same value (in the $j$-loop). It is usually eliminated by an optimizing compiler.
- A memory operation of type $\gamma$ is not affected by the DLR, because its operand cannot be reused due to the row-major matrix format assumption.
- A memory operation of type $\delta$ is affected by the DLR, due to its spatial locality.

*1) Evaluation of simplified RD model:* The number of cache misses during one execution of Example code 8 is denoted by $X$. The number of cache misses during one execution of Example code 8 with DLR($i \to j$) is denoted by $Y$. The reduction of the number of cache misses during one execution of Example code 8 due to the DLR($i \to j$) is denoted by $\mu_{\text{saved}}$ and it is equal to $X - Y$. The value of $\mu_{\text{saved}}$ has an upper bound

$$\mu_{\text{saved}} \leq (i_2 - i_1) \cdot DC_{\text{S}}/B_{\text{S}}.$$

This general upper bound can be reached only for loops where all memory operations are affected by the DLR. In practical cases, the reduction of the number of cache misses is smaller. To estimate the reduction of the number of cache misses during an execution of Example code 8 with the DLR, we need to count the number of iterations of the $j$-loop that can reside in the cache. We will denote this number by $N_{\text{iter}}$

$$N_{\text{iter}} = \frac{DC_{\text{S}}}{B_{\text{S}} \sum_m SCMO(m)}, \quad (1)$$

where

- $m$ is a memory operation (of type $\alpha - \delta$) in the $j$-loop,
- $SCMO(m)$ is the probability that memory operation $m$ loads data into a new cache block.

$$SCMO(m) = \begin{cases} 1 & \text{if } m \text{ is a memory operation of types } \beta \text{ or } \delta \text{ which are accessed in column-like pattern.} \\ S_{\text{D}}/B_{\text{S}} & \text{if } m \text{ is a memory operation of types } \alpha \text{ or } \gamma \text{ which are accessed in row-like pattern.} \end{cases} \quad (2)$$

If $N_{\text{iter}} < 1$, then the assumption (A5) is not satisfied and $\mu_{\text{saved}} = 0$.
If $N_{\text{iter}} \geq (j_2 - j_1)$, then the assumption (A6) is not satisfied and $\mu_{\text{saved}} = 0$.

We can also estimate probability (denoted by $PDLR(m)$) that the memory location accessed by memory operation $m$ is reused using DLR.

$$PDLR(m) = \begin{cases} 0 & \text{if } m \text{ is a memory operations of types } \beta \text{ or } \gamma \text{ (i.e., it is not affected by the DLR;)} \\ 1 - S_{\text{D}}/B_{\text{S}} & \text{if } m \text{ is a memory operation of type } \delta \text{ (i.e., it is affected by the DLR, for column-like access, the last element in cache-line is not counted;)} \\ 1 & \text{if } m \text{ is a memory operation of type } \alpha \text{ (i.e., it is affected by the DLR,for row-like access.)} \end{cases} \quad (3)$$

Finally, the number of cache misses saved by the DLR applied to the $i$-loop can be approximated by

$$\mu_{\text{saved}} = (i_2 - i_1) \cdot N_{\text{iter}} \sum_m \big(PDLR(m) \cdot SCMO(m)\big), \quad (4)$$

where $m$ is a memory operation in the $j$-loop.

Comparisons of the numbers of estimated and measured cache misses are presented in Section VI-C3.

## V. Experimental evaluation of the DLR

### A. Testing codes

For measuring of the effect of DLR (performance, cache miss rate, and so on), we use two simple codes:

- matrix-matrix multiplication (MMM for short),
- multiplication of two sparse matrices (spMMM for short).

We have deeply studied characteristics of these codes in following sections:

- For performance results, see Section VI-A.
- For cache utilization results, see Section VI-B.
- We also evaluate precision of our analytical model for MMM_STD code, see Section VI-C.
- We also combine effects of DLR and loop tiling for MMM_STD code, see Section VI-D.

*1) Matrix-matrix multiplication:* We consider input real square matrices $A$ and $B$ of order $n$. A standard sequential pseudocode for matrix-matrix multiplication $C = A \cdot B$ is the following:

```
1: procedure MMM_STD(in A,B;out C)
2:     for i ← 1,n do
3:         for j ← 1,n do
4:             sum = 0;
5:             for k ← 1,n do
6:                 sum+ = A[i][k] * B[k][j];
7:             C[i][j] = sum;
8:     return C;
```

*2) Multiplication of two sparse matrices:* We consider input real square sparse matrices $A$ and $B$ of order $n$ represented in the CSR format (see Section II-B), output matrix $C$ is a dense matrix of order $n$. A standard sequential pseudocode for the sparse matrix-matrix multiplication $C = A \cdot B$ can be described by the following pseudocode:

```
1: procedure SPMMM_CSR(in A,B;out C)
2:     for y ← 1,n do
3:         for i ← A.Addr[y], A.Addr[y+1] − 1 do
4:             x = A.Ci[i];
5:             for j ← B.Addr[x], B.Addr[x+1] − 1 do
6:                 x2 ← B.Ci[j];
7:                 C[y][x2]+ = A.Elem[i] * B.Elem[j];
8:     return C;
```

### B. Configuration of the experimental system

All cache events were evaluated by our software cache emulator [17] and verified by the Intel Vtune tool. The experiments were performed on the Pentium 4 Celeron at 2.4 GHz, 512 MB, running OS Windows XP Professional, with the following cache parameters:

- L1 data cache with $DC_S = 8K$, $B_S = 32$, $s = 4$, $h = 64$, and LRU strategy.
- L2 unified cache with $DC_S = 128K$, $B_S = 32$, $s = 4$, $h = 1024$, and LRU strategy.

We used the Intel compiler version 7.1 with switches:
`-O3 -fno_alias -xK -ipo`

## VI. The Results of Experimental Evaluation

### A. Performance evaluation of testing codes

We count every floating point operation (multiplication, addition and so on). The performance in MFLOPS is then defined as follows:

$$\text{MFLOPS(MMM\_STD)} = \frac{2n^3}{\text{execution time } [\mu\text{s}]}$$

$$\text{MFLOPS(SPMMM\_CSR)} = \frac{2 \cdot NZ_A \cdot NZ_B}{n \cdot \text{execution time } [\mu\text{s}]}$$

The graph in Figure 3 illustrates the performance with or without DLR. These graphs illustrate that the DLR increases the code performance due to better cache utilization. There

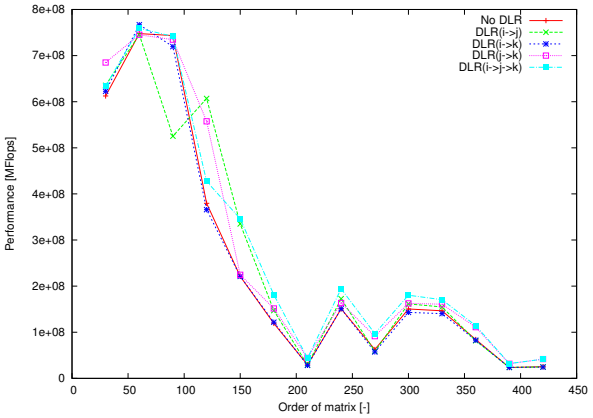


Fig. 3. Performance of MMM_STD.

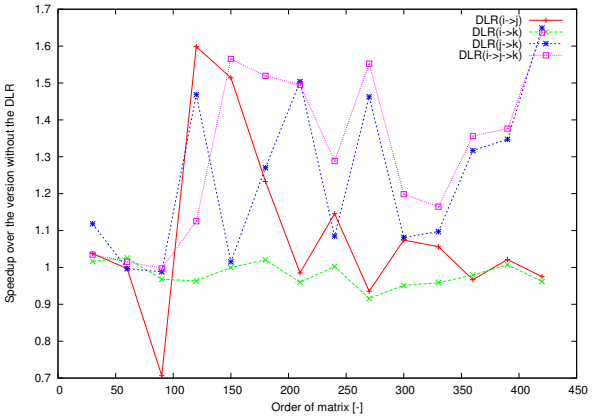


Fig. 4. Speedup of MMM_STD

is a performance gap (for example for $n = 120$ for the MMM_STD), which DLR can overcome. The graph in Figure 4 shows the speedup over the version without the DLR. We can conclude that the fastest code is the version with DLR($i \to j \to k$) for the MMM_STD code. We can also conclude that the average measured speedup is more than 20% in the measured set for the MMM_STD code.

For small matrices, a small slowdown was measured. While the DLR can improve the cache hit rate, it has more overhead due to more conditional loops. This effect becomes even more important for the DLR on triple loops.

### B. Cache miss rate evaluation

The cache utilization is enumerated according to the following definitions. Let us define "relative number of cache misses" as the ratio between the number of cache misses with DLR and the number of cache misses without DLR.

The graphs on Figures 5 and 6 illustrate the number of cache misses occurring during one execution of the MMM_STD pseudocode. We can conclude that

- the DLR effect depends on the value of the parameter $n$ and on the cache memory size (this observation proves the results of the analytical model from Section IV-B)

- except for few cases, the DLR transformation has a positive impact on cache utilization.

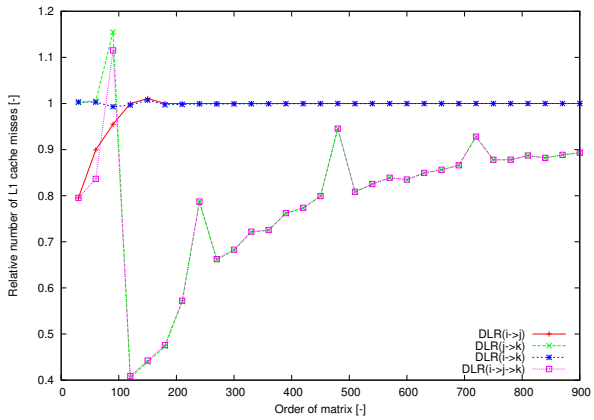


Fig. 5. Relative number of cache misses during MMM_STD for L1 cache

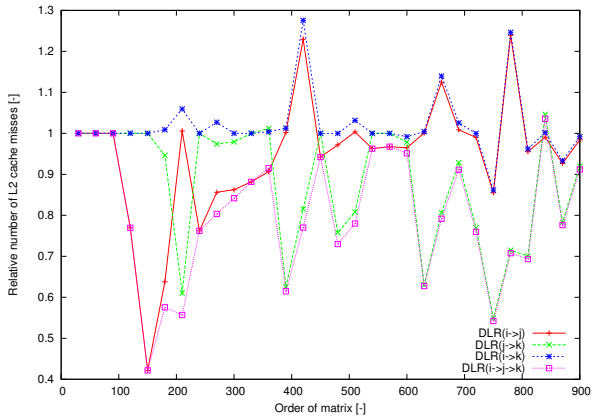


Fig. 6. Relative number of cache misses during MMM_STD for L2 cache

### C. Evaluation of simplified RD model

*1) Analytical cache model for* MMM_STD*:* To analyse this algorithm, we omit accesses in array $C$ at code line 7, because they are much less frequent. In this simplified model, the algorithm contains the following types of memory accesses:

- If DLR($i \to j$) is applied, then memory operations with $A[i][k]$ are of type $\beta$ and memory operations with $B[k][j]$ are of type $\alpha$.
- If DLR($j \to k$) is applied, then memory operations with $A[i][k]$ are of type $\alpha$ and memory operations with $B[k][j]$ are of type $\delta$.

*2) Analytical cache model for* SPMMM_CSR*:* Analysis of cache behaviour and DLR effects for this algorithm are beyond the scope of the compiler due to its irregular memory pattern.

*3) An example of evaluation of the cache analytical model:* We apply DLR($j \to k$) on the MMM_STD pseudocode. In this case as we stated above, memory operations with $A[i][k]$ are of type $\alpha$ and memory operations with $B[k][j]$ are of type $\delta$.

Firstly, we must count how many iterations of the $j$-loop can reside in the cache. From the types of memory operations (Eq. (2)), we can derive that

$$SCMO(A[i][k]) = S_{\mathrm{D}}/B_{\mathrm{S}}, \quad PDLR(A[i][k]) = 1.$$

$$SCMO(B[k][j]) = 1, \quad PDLR(B[k][j]) = 1 - S_{\mathrm{D}}/B_{\mathrm{S}}.$$

So, the number of iterations is (from cache parameters in Eq. (1)

$$N_{\mathrm{iter}} = \frac{DC_{\mathrm{S}}}{B_{\mathrm{S}}(1 + S_{\mathrm{D}}/B_{\mathrm{S}})}.$$

The number of cache misses saved by DLR$(k, j)$ per one iteration of the $j$-loop (Eq. (4)) is $\mu_{\mathrm{saved}} = N_{\mathrm{iter}}$.

The total number of cache misses saved by DLR($j \to k$) during one execution of the MMM_STD pseudocode is

$$\text{total } \mu_{\mathrm{saved}} = n^2 N_{\mathrm{iter}}.$$

For the given cache configuration, it gives the following results:

- for L1 cache: $N_{\mathrm{iter}} = 228$.
- for L2 cache: $N_{\mathrm{iter}} = 3640$.

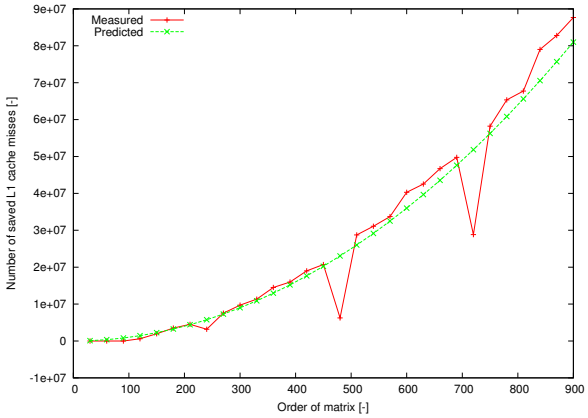


Fig. 7. Comparison of the numbers of estimated and measured cache misses ($\mu_{\mathrm{saved}}$) saved by the DLR during the execution of MMM_STD for L1 cache.

Comparisons of the numbers of estimated and measured cache misses are shown in Figures 7 and 8.

*4) Discussion of the precision of the simplified RD model:* Our analytical model is derived from the RD, which is based on fully-associative cache memory assumption. This assumption is the main source of errors in predictions. The errors are higher for L2 caches due to their lower associativity.

### D. Evaluation of combination of DLR and loop tiling

We have also measured the performance and cache utilization for pseudocode MMM_STD with loop tiling and effects of the DLR transformation on this code. Graphs on Figures 9 and 10 illustrate the fact that loop tiling can greatly improve the cache utilization. On the other hand, the tiling factor must be chosen very carefully, because the number of cache misses grows quickly with the distance of the tiling factor from the

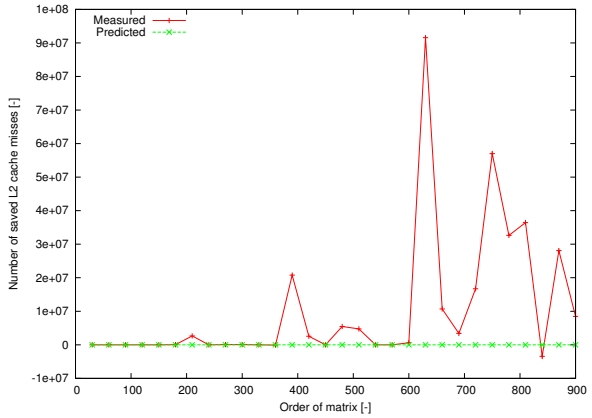


Fig. 8. Comparison of the numbers of estimated and measured cache misses ($\mu_{\text{saved}}$) saved by the DLR during the execution of MMM_STD for L2 cache.

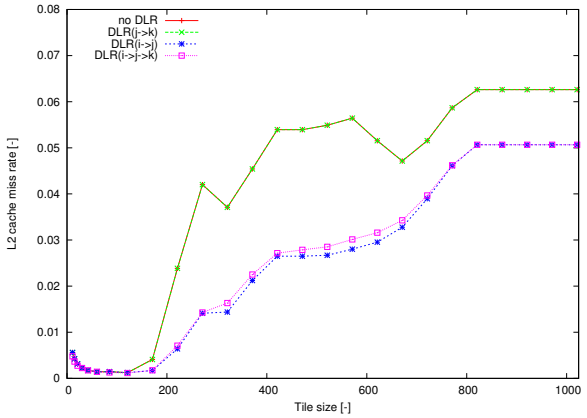


Fig. 10. The L2 miss rate for MMM_STD code with loop tiling (for $n$=1024) for different values of the tile size.

optimal value. When the DLR is applied, the growth is more smooth, so the code is less sensitive to the tiling factor value. Hence, the DLR technique is useful in cases when it is hard to predict a good value for the tiling factor.

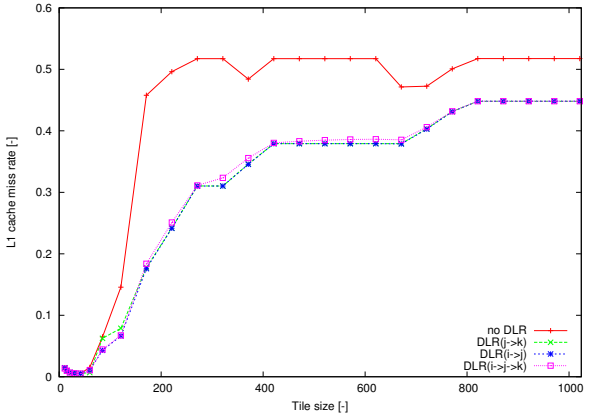


Fig. 9. The L1 miss rate for MMM_STD code with loop tiling (for $n$=1024) for different values of the tile size.

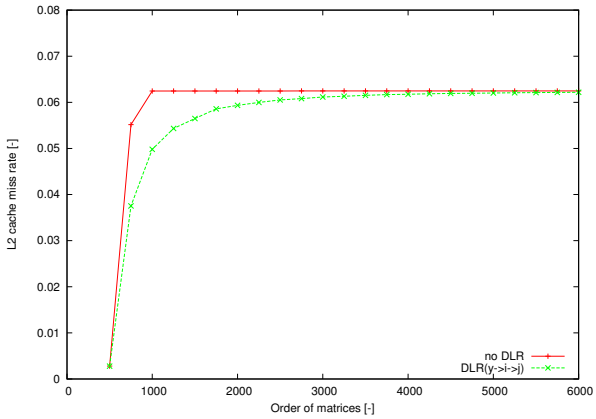


Fig. 11. The L1 and L2 miss rate for SPMMM_CSR algorithm (for $density(A) = 7\%$ and $density(B) = 21\%$)

### E. Evaluation of the DLR for the SPMMM_CSR code

The SPMMM_CSR code is a simple example of an irregular code. For the testing purposes, we always generate five sparse matrices with random locations of nonzero elements with given properties (order of matrix, number of nonzero elements or density). The average value of these five measurements were taken as a result. In this code, the memory access pattern is hard to predict on the compiler level and loop tiling is excluded. Thus the DLR is usable and the application of this technique can save reasonably large number of cache misses (see Figures 11,12, and 13).

## VII. Automatic compiler support of the DLR

The DLR transformation brings new possibilities to optimize nested loops.

### A. A proposed algorithm of automatic compiler support of the DLR

Let $\mathcal{L}_{1\ldots b}$ represent a hierarchy of immediately nested loops ($\mathcal{L}_1$ is the outermost loop, $\mathcal{L}_b$ is the innermost loop). The control variable for the loop $\mathcal{L}_i$ is denoted by $\mathcal{C}_i$. We propose the following function that returns a list of loop numbers that can profit from the DLR application and that can be implemented into compiler to support the DLR application automatically.

1: **procedure** DLR_APPLICATION(in $b, \mathcal{L}, \mathcal{C}$)
2: $res = []$;
3: **for** $i \leftarrow 1, b-1$ **do** ▷ here we consider application of DLR($\mathcal{C}_i \rightarrow \mathcal{C}_{i+1}$)
4: **if** this DLR application is possible **then**
5: compute $\mu_{\text{saved}}$ from the proposed cache model;
6: compute overhead of this DLR application;
7: **if** this DLR application pays-off **then**
8: add $i$ to the $res$;
9: **return** $res$;

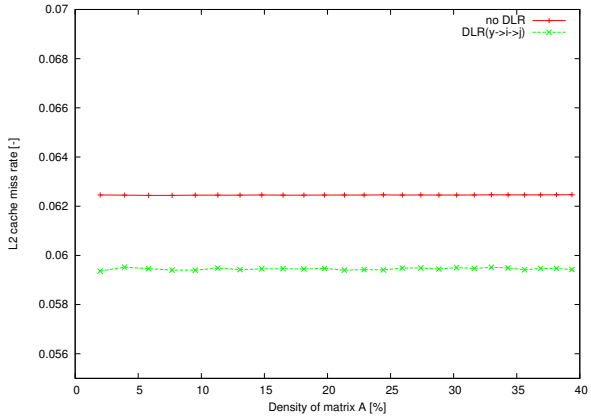


Fig. 12. The L2 miss rate for algorithm SPMMM_CSR ($n$=2200 and $density(B) = 17\%$)

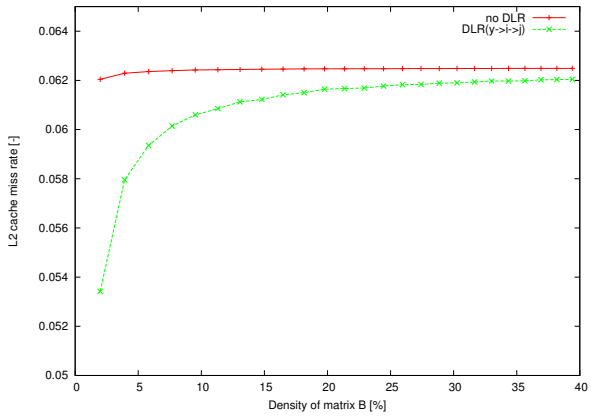


Fig. 13. The L2 miss rate for algorithm SPMMM_CSR ($n$=3700 and $density(A) = 10\%$)

If this function returns empty list, then DLR does not pay-off for any loop in $\mathcal{L}$. In other case, it returns a list ($res$) of loop number $x$ such that the DLR should be applied to $\mathcal{L}_x$, i.e., DLR($\mathcal{C}_x \rightarrow \mathcal{C}_{x+1}$) to increase the code performance.

### B. Discussion of applicability of the DLR inside compilers

The function in Section VII-A is very general. The real incorporation of the DLR into existing compilers (like GCC or LLVM) must address more issues:

- Where can be the DLR applied? The DLR can be applied on the nested reversible loops. This condition can be easily checked by the compiler.
- Where should be DLR applied? The DLR should be applied on a pair or triple of loops that causes its maximal effect (mentioned in Section III-C). This compiler decision is very similar as for loop tiling.
- Has DLR significant effect? Yes. In most cases, higher speedups are achieved by loop unrolling or loop tiling. But the DLR can be combined with these techniques (see Section VI-D) and also the DLR can be applied on some codes where loop tiling could not (for example sparse matrix operations).

## VIII. Conclusions

We have described a new code transformation technique, the dynamic loop reversal, whose goal is to improve temporal locality. This transformation seems to be very useful for codes with nested loops. We have demonstrated significant performance gains for two basic algorithms from linear algebra.

We have also developed a probabilistic analytical model for this transformation and compared the numbers of measured cache misses and the numbers of cache misses estimated by the model. The inaccuracies of the model are due to some simplifying assumptions.

This work is to contribute to the development of more efficient compiler techniques.

## References

[1] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] K. R. Wadleigh and I. L. Crawford, *Software optimization for high performance computing*. Hewlett-Packard professional books, 2000.

[3] M. Wolfe, *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[4] X. Vera and J. Xue, "Efficient compile-time analysis of cache behaviour for programs with IF statements," Beijing, October 2002. [Online]. Available: citeseer.ist.psu.edu/567600.html

[5] N. Ahmed, N. Mateev, and K. Pingali, "Tiling imperfectly-nested loop nests," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000, p. 31.

[6] J. Xue, *Loop tiling for parallelism*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[7] P. Tvrdík and I. Šimeček, "Analytical model for analysis of cache behavior during cholesky factorization and its variants," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPP 2004)*, vol. 12, Montreal, Canada, 2004, pp. 190–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018426.1020360

[8] E. Im, *Optimizing the Performance of Sparse Matrix-Vector Multiplication - dissertation thesis*. University of Carolina at Berkeley: Dissertation thesis, 2001.

[9] D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Modeling data locality for the sparse matrix-vector product using distance measures," *Parallel Computing*, vol. 27, no. 7, pp. 897–912, Jun. 2001.

[10] P. Tvrdík and I. Šimeček, "Analytical modeling of optimized sparse linear code," in *Parallel Processing and Applied Mathematics*, vol. 3019/2004, no. 4, Czestochova, Poland, 2003, pp. 207–216. [Online]. Available: http://www.springerlink.com/content/drwdhen7db199k05/

[11] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 252–262.

[12] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, pp. 30–44, May 1991. [Online]. Available: http://doi.acm.org/10.1145/113446.113449

[13] S. Carr and R. Lehoucq, "Compiler blockability of dense matrix factorizations," *ACM Transactions on Mathematical Software*, vol. 23, pp. 336–361, 1996.

[14] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *SIGPLAN Not.*, vol. 34, pp. 215–228, May 1999. [Online]. Available: http://doi.acm.org/10.1145/301631.301668

[15] X. Vera and J. Xue, "Let's study whole-program cache behaviour analytically," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, ser. HPCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 175–. [Online]. Available: http://dl.acm.org/citation.cfm?id=874076.876456

[16] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of PDCS'01*, August 2001, pp. 617–662. [Online]. Available: citeseer.ist.psu.edu/beyls01reuse.html

[17] P. Tvrdík and I. Šimeček, "Software cache analyzer," in *Proceedings of CTU Workshop*, vol. 9, Prague, Czech Republic, Mar. 2005, pp. 180–181.