Computer Science and Information Systems 11(4):1381–1416 DOI: 10.2298/CSIS131126075S

A new code transformation technique for nested loops

Ivan Šimeček[1] and Pavel Tvrdík[1]

Department of Computer Systems,
Faculty of Information Technology,
Czech Technical University in Prague, 160 00,
Prague, Czech Republic
{xsimecek,pavel.tvrdik}@fit.cvut.cz

Abstract. For good performance of every computer program, good cache utilization is crucial. In numerical linear algebra libraries, good cache utilization is achieved by explicit *loop restructuring* (mainly *loop blocking*), but it requires a complicated memory pattern behavior analysis. 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 cache behavior and evaluate results of the model accuracy by the 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 the model is relatively accurate and can be used in practice.

Keywords: Source code optimization, loop transformations, model of cache behavior.

1. Introduction

1.1. General

Due to the increasing difference between memory and processor speed, it becomes critical to minimize communications between main memory and processor. It is done by addition of cache memories on the data path. The memory subsystem of nowadays processors is organized as a hierarchy in which the latency of memory accesses increases rapidly from one level of hierarchy to the next. Numerical codes (e.g. libraries such as BLAS [9] or LAPACK [2]) are now some of the most demanding programs in terms of execution time and memory usage. These numerical codes consist mainly of loops. Programmers and optimizing compilers often restructure a code to improve its cache utilization. Many source code transformation techniques have been developed and used in the state-of-the-art optimizing compilers. In this paper, we consider the following standard techniques: *loop*

This is a revised and extended version of the paper that appeared in proceedings of WAPL'13 [13]. We included a more in-depth analysis of the effects of the dynamic loop reversal, presented two possible implementation variants and reduced some limitations of the cache model. The original paper was strictly focused on the codes from numerical linear algebra, in this paper we discuss also the codes from graph theory etc.

unrolling, *loop blocking*, *loop fusion*, and *loop reversal* [15, 30, 32, 8]. Models for predicting the number of cache misses have also been developed to have detailed knowledge about the impacts of these transformation techniques on the cache utilization. The existing literature aimed at the maximal cache utilization within numerical codes focused mainly on regular memory accesses, but there is also an important subset of numerical codes where memory accesses are irregular (e.g. for computations with sparse matrices). As far as we know, there is no transformation technique that can improve the cache behavior for regular and irregular codes.

The main result of this paper is a description of a new transformation technique, called *dynamic loop reversal*, shortly *DLR*. This transformation technique can lead to better cache utilization (even for codes with irregular memory accesses) due to the improved temporal and spatial locality with no need to analyze the memory pattern behavior.

In Section 2, we introduce the basic terminology (the used model of the cache architecture and the basic sparse matrix storage format). In Section 3, we briefly discuss the motivation that lead us to the design of DLR transformation. In order to incorporate the DLR into compilers, we propose models for predicting the number of cache misses for DLR in Section 4. In Section 5, we describe experiment's settings and configurations used for measurement the effects of DLR. In Section 6, we measure and evaluate the effects of DLR (e.g. performance, cache miss rate) on example codes. In Section 7, we discuss an idea of an automatic compiler support of DLR to optimize nested loops.

1.2. Related works

There are many existing source code transformation techniques (for details see [15, 30, 32]). Models for enumeration of the number of cache misses have also been developed see ([27, 1, 33]) for predicting the impacts of these transformation techniques on the cache utilization. But only few of existing papers also aim at codes with irregular memory accesses (e.g. computations with sparse matrices). In [8], optimizations for multicore stencil (nearest-neighbor) computations are explored. This class of algorithms is used in many PDE solvers. Authors develop a number of effective optimization strategies and propose an auto-tuning environment.

The other approach for optimization of multicore stencil computations is proposed in [21]: programmer should specify a stencil in a domain-specific stencil language [17] — Pochoir language. The resulting optimizations are derived from this specification.

There are also many algorithms and storage formats designed for the acceleration of the sparse matrix-vector multiplication (e.g. [29, 11, 25, 12]).

But all mentioned papers are focused only on the specific operation, the proposed DLR transformation is more general and can be applied to non-specific codes.

2. Terminology

In the paper, we assume that all elements of vectors and matrices are of the type *double* and that all matrices are stored in a row-major format.

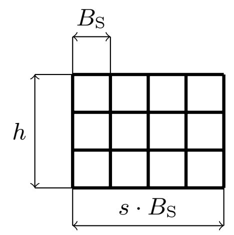

Fig. 1. Description of the cache parameters.

2.1. The cache architecture model

We consider a most used model of the *set-associative cache* data cache (for details see [14]). The number of sets is denoted as h (see Figure 1). One set consists of s independent *blocks*. The size of the data cache in bytes is denoted as DC_S. The cache block size in bytes is denoted as B_S. Then $DC_S = s \cdot B_S \cdot h$. The size of the type `double` is denoted as S_D. We consider only *write-back* caches with an *least recently used (LRU) block replacement* strategy.

We consider two types of cache misses:

- *Compulsory* misses (sometimes called *intrinsic* or *cold*) that occur when new data is loaded into empty cache blocks.
- *Thrashing* misses (also called *cross-interference*, *conflict*, or *capacity* misses) that occur because the useful data was replaced prematurely from the cache block.

In modern architectures, there are three level of caches, but we can consider each level independently. Some caches are unified (for data and instructions), but we assume them as data caches because the size of (most frequent) code is negligible.

2.2. Compressed sparse row (CSR) format

A matrix A of order n is considered to be *sparse* if it contains much less nonzero elements than n^2 otherwise it is considered as *dense*. Some alternative definitions of sparse matrix can be found in [22]. The most common format (see [10, 11, 23, 25]) for storing sparse matrices is the *compressed sparse row* (CSR) format. The number of nonzero elements is denoted as NZ_A. A matrix A stored in the CSR format is represented by three linear arrays $Elem_A$, $Addr_A$, and Ci_A (see Figure 2 b). Array $Elem_A[1, \ldots, NZ_A]$ stores the nonzero elements of A. Array $Addr_A[1, \ldots, n+1]$ contains indexes of initial nonzero elements of rows of A. Hence, the first nonzero element of row j is stored at index $Addr_A[j]$ in array $Elem_A$, obviously $Addr_A[1] = 1$ and $Addr_A[n+1] = NZ_A$. Array $Ci_A[1, \ldots, NZ_A]$ contains column indexes of nonzero elements of A. The density of a matrix A (denoted as $density(A)$) is the ratio between NZ_A and n^2.

3. Code restructuring

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

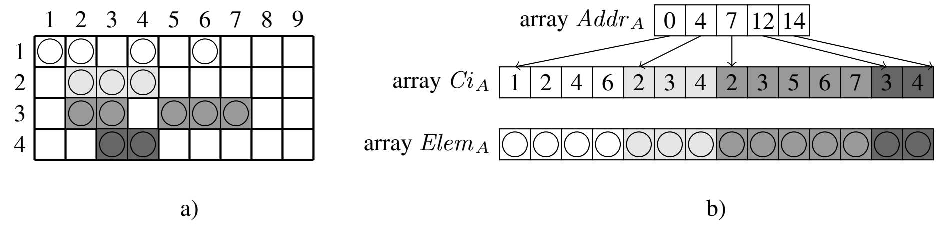

Fig. 2. a) Example of a sparse matrix, b) Representation of this matrix in the CSR format.

3.1. Standard static loop reversal

In standard loop reversal, the direction 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 (e.g. loop unrolling or loop blocking) in general.

Example code 1
```
1: for $i \leftarrow n, 2$ do
2:     $B[i] \leftarrow B[i] + B[i-1]$;
3: for $i \leftarrow 2, n$ do
4:     $A[i] \leftarrow A[i] + B[i]$;
```

Example code 1 represents a typical combination of data-dependent loops whose data dependency can be recognized automatically by common compiler optimization techniques. However, both loops are *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 (for the definition of the reuse distance see Section 4.1) for the memory transactions on array B are decreased.

Example code 2 Loop reversal and loop fusion applied to Example code 1
```
1: for $i \leftarrow n, 2$ do
2:     $B[i] \leftarrow B[i] + B[i-1]$;
3:     $A[i] \leftarrow A[i] + B[i]$;
```

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

Example code 3

```
1: for $i \leftarrow 1, n$ do                                ▷ Loop 1
2:     $s \leftarrow s + A[i] * A[i]$;
3: $norm \leftarrow \sqrt{s}$;
4: for $i \leftarrow 1, n$ do                                ▷ Loop 2
5:     $A[i] \leftarrow A[i]/norm$;
```

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 $i \leftarrow 1, n$ do                                ▷ Loop 1
2:     $s \leftarrow s + A[i] * A[i]$;
3: $norm \leftarrow \sqrt{s}$;
4: for $i \leftarrow n, 1$ do                                ▷ Loop 2
5:     $A[i] \leftarrow A[i]/norm$;
```

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. This idea was used in [23] for the acceleration of Conjugate Gradient method.

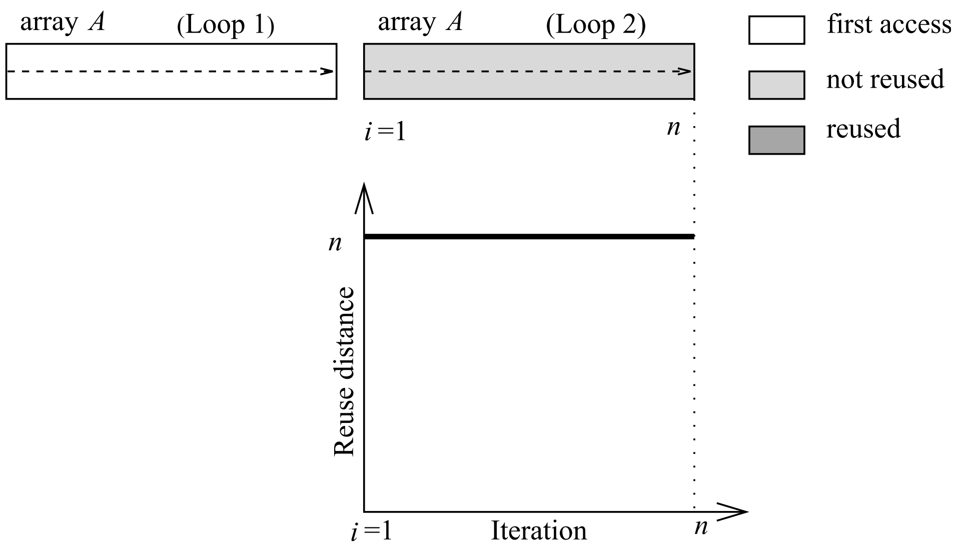

Fig. 3. Reuse distances in *Example code 3*.

3.2. Effect of the static loop reversal on cache behavior

If the size of array A is less than the cache size ($nS_{\mathrm{D}} \leq DC_{\mathrm{S}}$), then Example code 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. Thus, loop reversal improves the temporal locality (see Figures 3 and 4 for the comparison of reuse distances).

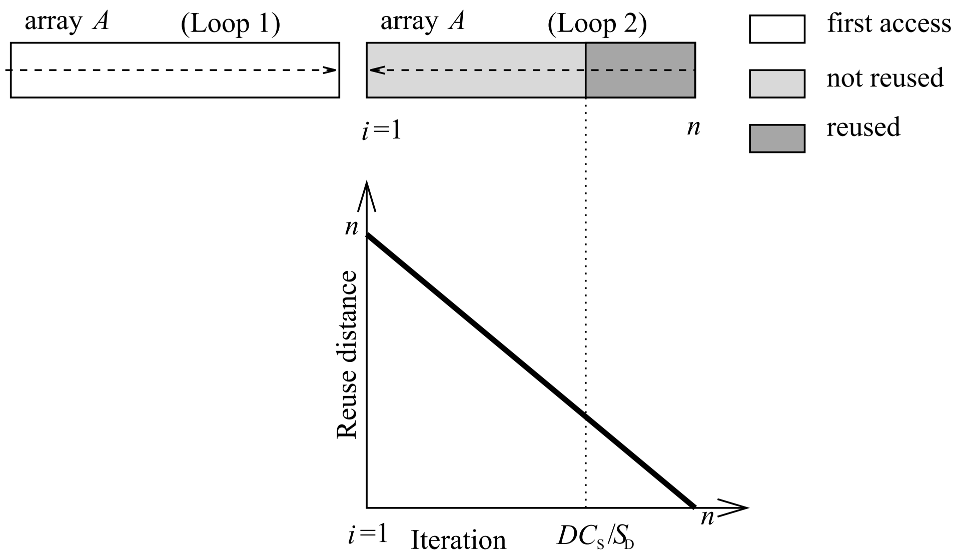

Fig. 4. Reuse distances in *Example code 4* (*Example code 3* after loop reversal).

3.3. Dynamic loop reversal

The static loop reversal is used to reverse data-dependency in one dimension. This motivated us to generalize this idea, and therefore we designed another optimization for nested reversible loops based on loop reversal. Consider the following code (Example code 5):

Example code 5

```
1: for $i \leftarrow 1, n$ do
2:     $s \leftarrow 0$;
3:     for $j \leftarrow 1, n$ do
4:         $s \leftarrow s + A[i][j] * x[j]$;
```

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 the 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

```
for $i \leftarrow 1, n$ do
    $s \leftarrow 0$;
    if $i$ is odd then
        for $j \leftarrow 1, n$ do
            $s \leftarrow s + A[i][j] * x[j]$;
    else
        for $j \leftarrow n, 1$ do
            $s \leftarrow s + A[i][j] * x[j]$;
```

This transformation will be denoted as DLR($i \to j$). For large arrays, this transformation (the result is Example code 6) leads to even better temporal locality than the original Example code 5, because it reduces the reuse distances, and some data resides 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 $DC_{\mathrm{S}}/B_{\mathrm{S}}$. Therefore, *DLR* has a significant effect if the cache size is comparable to the sum of the affected array sizes in one iteration of the outer loop. The necessary condition for applying *DLR* is that the inner loop must be reversible.

Possible implementations of DLR The straightforward implementation, described above by the code "DLR applied to Example code 5"(Example code 6), leads to an overhead due to one additional conditional jump that is hard to predict due to its pattern. We call it "implementation variant A".

We propose the other possible implementation as follows.

Example code 7 DLR applied to Example code 5

```
for $i \leftarrow 1, n$ step 2 do
    $s \leftarrow 0$;
    for $j \leftarrow 1, n$ do
        $s \leftarrow s + A[i][j] * x[j]$;
    for $j \leftarrow n, 1$ do
        $s \leftarrow s + A[i+1][j] * x[j]$;
if $n$ is odd then
    for $j \leftarrow 1, n$ do
        $s \leftarrow s + A[n][j] * x[j]$;
```

The second implementation (Example code 7) leads to a faster code, since the conditional jump inside the i-loop is eliminated, but in fact, it is based on the loop unrolling. We call it "implementation variant B".

Possible interference of DLR with heuristics in optimizing compilers On the other hand, DLR may have a disadvantage due to the fact that it can confuse optimizing compilers and hinder further loop optimizations. For example, in Example code 5, there are two nested loops and after the application of DLR (implementation variant A), the code contains also a conditional branch that is non-trivial for the compiler heuristics (see [33]).

The application of DLR on triple-nested loops In the previous text, DLR was applied to double-nested loops, but it can also be applied to triple-nested loops. Consider the following code skeleton (Example code 8):

Example code 8

```
for $i \leftarrow 1, n$ do
    for $j \leftarrow 1, n$ do
        for $k \leftarrow 1, n$ do
            (* loop body *)
```

In Example code 8, 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 lies in the composition of two transformations DLR($i \to j$) and DLR($j \to k$). This composition we will denoted as DLR($i \to j \to k$). In this case, the effect of DLR is twofold: DLR($i \to j$) (on the outer pair of the loops) can improve a temporal locality inside the L2 cache and DLR($j \to k$) (on the inner pair of the loops) can improve a temporal locality inside the L1 cache.

3.4. Comparison and possible combinations of DLR and other loop restructuring techniques

Loop unrolling Loop unrolling has two main effects. Firstly, it makes the sequential code longer, so it may improve the 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 behavior.

If the straightforward implementation of DLR (variant A, see Section 3.3) is used, the loop unrolling and DLR can be fully combined. If the second implementation (variant B, see Section 3.3) is used, the loop unrolling is an essential part of a DLR implementation. Hence it is a natural combination of both techniques, but the loop unrolling factor must be even.

Loop tiling (blocking) Loop tiling (sometimes called loop blocking or iteration space tiling) is one of the advanced loop restructuring techniques (for details see [7, 31, 5, 6, 20, 28]). The compiler can use it to increase the cache hit rate. The following code (Example code 9) is an example of an application of the loop tiling technique.

Example code 9

```
for $i_1 \leftarrow 1, n$ step $B_f$ do
    for $j_1 \leftarrow 1, n$ step $B_f$ do
        for $k_1 \leftarrow 1, n$ step $B_f$ do
            for $i \leftarrow i_1, \min(i_1 + B_f - 1, n)$ do
                for $j \leftarrow j_1, \min(j_1 + B_f - 1, n)$ do
                    for $k \leftarrow k_1, \min(k_1 + B_f - 1, n)$ do
                        $C[i][j] \leftarrow C[i][j] + A[i][k] * B[k][j];$
```

One possible motivation for using this technique is the fact that the *loop range* (i.e., the size of the array traversed repeatedly within the loop) is too big and exceeds the data cache size DC_{S}. Thus, the loop should be split into two loops: the outer loop (out-cache loop) and the inner loop (in-cache loop). The value B_f is called a *tiling or block factor*, whose its optimal value depends on the cache size.

The loop tiling and DLR can be easily combined. DLR can be applied on every pair of immediately nested loop, but it is useless to apply it for in-cache loops (i-loop, j-loop, and k-loop). We consider loop tiling a competitor to DLR and thus we performed experiments with both. These quantitative measurements of the effects of these techniques are presented in Section 6.4.

4. Analytical model of the cache behavior for DLR

To estimate parameters for loop restructuring techniques, modern compilers use the *polytope model* [32, 33]. We will present three cache behavior models based on reuse distances (shortly RD) that can be derived from the polytope model.

4.1. Cache miss model with reuse distances

This model is inspired by the model introduced in [4, 3, 24]. We will call it *basic RD model*.

Definition 1. *Consider an execution of an algorithm on the computer with load/store architecture and assume that the addresses of memory transactions 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* $t \in (1, n]$*, 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* $RD(t) = \infty$*.*

Obviously, in the above definition, $RD(t) \leq \epsilon(t)$ or $RD(t) = \infty$. 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 a new value and a cache miss occurs. If $RD(t) = \infty$, then a compulsory miss occurs, otherwise a thrashing miss occurs. Remember 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 create a more realistic model.

4.2. The cache miss model with generalized reuse distances

To address this drawback of the basic RD model, we define a more general form of reuse distances. We call this model *generalized RD model*.

Definition 2. *Consider a* sequence of memory access addresses $P[1, \ldots, n]$. *Address* $addr_i$ *from* P *is mapped onto a cache block marked by tag* tag_i *computed by the corresponding cache memory mapping function. Then* $P'[1, \ldots, n] = [tag_1, \ldots, tag_n]$ *is called a* **sequence of cache memory tags**. *Then the* **generalized reuse distance** $GRD(t)$*, where* $t \in (1, n]$*, is defined as the number of* **different** *cache blocks accessed between two uses of the tag* $P'[t]$. *Formally, if* $P'[t] = tag_t$ *and* $\epsilon(t) > 0$ *is the minimal integer number such that* $P'[t - \epsilon(t)] = tag_t$*, then* $GRD(t) = |P'[t - \epsilon(t)], \ldots, P'[t - 1]|$. *If such an* $\epsilon(t)$ *does not exist, then* $GRD(t) = \infty$.

The generalized reuse distances can be useful for estimating the numbers of cache misses. If $GRD(t) > DC_{\mathrm{S}}/B_{\mathrm{S}}$, then the content of cache block $P'(t)$ is replaced by a new value, and a cache miss occurs (if $GRD(t) = \infty$, then a compulsory miss occurs; otherwise, a thrashing miss occurs).

Both cache miss models based on the reuse distances (RD and GRD) has several drawbacks:

- Reuse distances for a given memory address or cache memory tag vary in time. Intervals of reuses are not the same during the execution ($RD(t_1) \neq RD(t_2)$ for $P[t_1] = P[t_2]$).
- The mapping function of the cache memory is not considered. Essentially, the cache is assumed fully-associative ($h = 1$).

4.3. Simplified cache miss model for DLR

Even the basic RD model is too complicated for modeling the cache behavior of DLR in real applications. Hence, we introduce the other model that is even more simplified. The model will be called *simplified RD model*. We use this model for the enumeration of cache misses saved by DLR. To derive an analytical model of a DLR effect on the cache behavior, consider the following code skeleton (Example code 10) representing most frequent memory access patterns during a matrix computation (k, l are the small constant, $k, l \in \mathcal{N}, k > 0$):

Example code 10

```
statement$_1$;
for $i \leftarrow i_1, i_2$ do
    statement$_2$;
    for $j \leftarrow j_1, j_2$ do
        statement$_3$;
        $\cdots \leftarrow B[k*j+l]$;          ▷ Memory operation of type $\alpha$
        $\cdots \leftarrow B[k*i+l]$;          ▷ Memory operation of type $\beta$
        $\cdots \leftarrow A[i][k*j+l]$;       ▷ Memory operation of type $\gamma$
        $\cdots \leftarrow A[j][k*i+l]$;       ▷ Memory operation of type $\delta$
    statement$_4$;
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 zero cache effects and the only memory accesses are memory operations of type $\alpha - \delta$.
A3 We assume that the generalized reuse distances depend on the exact ordering of memory operations (inside the j-loop) only slightly and so does 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, DLR has no effect in comparison to standard execution.
A7 This model is derived only for immediately nested loops.
A7 The size of the constant l does not have any impact on the generalized reuse distances.

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

– A memory operation of type α is affected by DLR, because its operand (or its part) can be reused. The effect of DLR can be estimated by an RD or GRD analysis.
– A memory operation of type β is not affected by DLR, because it returns the same value (in the j-loop). It is usually eliminated by an optimizing compiler.
– A memory operation of type γ is not affected by DLR, because its operand cannot be reused due to the row-major matrix format assumption.
– A memory operation of type δ is affected by DLR due to its spatial locality. But it cannot be estimated by an RD analysis defined in Section 4.1. Instead, a GRD analysis must be applied, as defined in Section 4.2.

Evaluation of simplified RD model The number of cache misses during one execution of *Example code 5* is denoted as X. The number of cache misses during one execution of *Example code 5* with DLR($i \to j$) is denoted as Y. The reduction of the number of cache misses during one execution of Example code 5 due to the DLR($i \to j$) is denoted as μ_{saved}) and it is equal to $X - Y$. The value of μ_{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 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 5* with DLR, we need to count the number of iterations of the j-loop that can reside in the cache. We will denote this number as N_{iter}

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

where

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

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

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 the probability (denoted as $PDLR(m)$) that the memory location accessed by a memory operation m is reused using DLR.

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

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

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

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

Comparisons of the numbers of estimated and measured cache misses are presented in Section 6.3.

5. Experimental evaluation of the DLR

5.1. Example codes

For measuring the effect of DLR (performance, cache miss rate, etc.), we implement the following six simple codes in C/C++ (the implementations are inspired by [18]):

– matrix-matrix multiplication (MMM for short),
– Cholesky factorization (CHF for short),
– multiplication of two sparse matrices (spMMM for short),
– Knapsack problem (KP for short),
– Floyd-Warshall algorithm (FW for short),
– Maximum flow algorithm (MF for short).

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

– For performance results, see Section 6.1.
– For cache utilization results, see Section 6.2.
– We also evaluate precision of our analytical model for `MMM_STD` code, see Section 6.3.
– We also combine effects of DLR and loop tiling for `MMM_STD` code, see Section 6.4.
– For cache utilization results of `spMMM_CSR` code, see Section 6.5.

Matrix-matrix multiplication We consider input real matrices A and B of the 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 $i \leftarrow 1, n$ do
3:         for $j \leftarrow 1, n$ do
4:             $sum \leftarrow 0$;
5:             for $k \leftarrow 1, n$ do
6:                 $sum \leftarrow sum + A[i][k] * B[k][j]$;
7:             $C[i][j] \leftarrow sum$;
8:     return $C$;
```

Cholesky factorization Let A be a symmetric dense matrix of the order n. The task of the Cholesky factorization is to compute a lower triangular matrix L of the order n such that $A = LL^T$. A standard algorithm for the Cholesky factorization is described by the following pseudocode.

```
1: procedure CHF_STD(in $A$;out $L$) ▷ The input matrix $A$ is overwritten by the values
   of a matrix $L$
2:     for $j \leftarrow 1, n$ do
3:         $sum \leftarrow 0$;
4:         for $l \leftarrow 1, j-1$ do
5:             $sum \leftarrow sum + (A[j][l])^2$;
6:         $A[j][j] \leftarrow \sqrt{A[j][j] - sum}$;
7:         for $i \leftarrow j+1, n$ do
8:             $sum \leftarrow 0$;
9:             for $k \leftarrow 1, j-1$ do
10:                $sum \leftarrow sum + A[i][k] * A[j][k]$;
11:            $A[i][j] \leftarrow \frac{A[i][j]-sum}{A[j][j]}$;
12:    $L \leftarrow A$;
13:    return $L$;
```

Multiplication of two sparse matrices We consider input real square sparse matrices A and B of the order n represented in the CSR format (see Section 2.2), output matrix C is a dense matrix of the 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 $y \leftarrow 1, n$ do
3:         for $i \leftarrow A.Addr[y], A.Addr[y+1]-1$ do
4:             $x \leftarrow A.Ci[i]$;
5:             for $j \leftarrow B.Addr[x], B.Addr[x+1]-1$ do
6:                 $x2 \leftarrow B.Ci[j]$;
7:                 $C[y][x2] \leftarrow C[y][x2] + A.Elem[i] * B.Elem[j]$;
8:     return $C$;
```

Knapsack problem Let there be n items (i_1 to i_n), where i_k has a value v_k and weight w_k. The maximum weight that can be carried in the bag is W. A standard sequential pseudocode for Knapsack problem using the dynamic programming follows:

```
1: procedure KP_STD(in $n$,$v$,$w$,$W$; out $res$)
2:     for $i \leftarrow 1, W$ do
3:         $t_{old}[i] \leftarrow 0$;
4:     for $i \leftarrow 1, n$ do
5:         for $j \leftarrow 1, W$ do
6:             if $j \geq w[i]$ then
7:                 $t_{new}[j] \leftarrow \max(t_{old}[j], t_{old}[j - w[i]] + v[i])$;
8:             else
9:                 $t_{new}[j] \leftarrow t_{old}[j]$;
10:        exchange $t_{new}$ and $t_{old}$;
11:    $res \leftarrow t_{old}[W]$;
12:    return $res$;
```

Floyd-Warshall algorithm We consider input graph $G = (V, E)$. A standard sequential pseudocode for the Floyd-Warshall algorithm follows:

```
1: procedure FW_STD(in $V$,$E$; out $dist$)
2:     for $i \leftarrow 1, |V|$ do
3:         for $j \leftarrow 1, |V|$ do
4:             $dist[i][j] \leftarrow \infty$;
5:     for $i \leftarrow 1, |V|$ do
6:         $dist[i][i] \leftarrow 0$;
7:     for $i \leftarrow 1, |V|$ do
8:         $(u, v) \leftarrow V_i$;
9:         $dist[u][v] \leftarrow w(u, v)$;                ▷ the weight of the edge $(u, v)$
10:    for $k \leftarrow 1, |V|$ do
11:        for $i \leftarrow 1, |V|$ do
12:            for $j \leftarrow 1, |V|$ do
13:                $dist[i][j] \leftarrow \min(dist[i][j], dist[i][k] + dist[k][j])$;
14:    return $dist$;
```

Maximum flow algorithm We consider an input network $G = ((V, E), c, s, t)$. A standard sequential pseudocode for the Dinic's blocking flow algorithm can be described by the following pseudocode:

```
1: procedure MF_STD(in $G$;out $f$)
2:     loop
3:         Construct $G_L$ from the residual graph $G_f$ of $G$.
4:         if $\nexists$ path from $s$ to $t$ in $G_L$ then
5:             break;
6:         Find a blocking flow $f'$ in $G_L$
7:         Augment flow $f$ by $f'$
8:     return $f$;
```

5.2. Hardware and software configuration of the experimental system

All cache events were evaluated by our software cache emulator [26] and verified by the Intel Vtune or Cachegrind (a part of Valgrind [19]) tool. All DLR transformations are done manually or using a simple wrapper on the source code level. In all codes, we used the DLR implementation variant A (see Section 3.3). Implementation variant A is a "pure"application of DLR, implementation variant B is a composition of DLR and loop unrolling. So, for measurement of DLR effects only, implementation variant A is more suitable.

The measurements of `MMM_STD` and `CHF_STD` are straightforward. But for the other codes, the results depends not only on the problem size but on the exact structure of input data (e.g. graph for `MF_STD` code). So, it's not possible to show results for all combinations of input data.

– `spMMM_CSR` code: For the testing purposes, we always generate five sparse matrices with random locations of nonzero elements of given properties (the order of a matrix, the number of nonzero elements or density). The average value of these five measurements were taken as a result.

– `KP_STD` code: For the testing purposes, we always generate five inputs with same value of n with random values of v and w. The average value of these five measurements were taken as a result.
– `FW_STD` code: For the testing purposes, we always generate five random graphs with same value of $|V|$. The average value of these five measurements were taken as a result.
– `MF_STD` code: For the testing purposes, we always generate five random graphs with same values of $|V|$ and $|E|$. The average value of these five measurements were taken as a result.

Testing configuration 1

Some experiments were performed on the Pentium 4 Celeron at 2.4GHz, 512 MB, running OS Windows XP Professional, with the following cache parameters:

- L1 data cache with $DC_\mathrm{S} = 8K$, $B_\mathrm{S} = 32$, $s = 4$, $h = 64$, and LRU strategy.
- L2 unified cache with $DC_\mathrm{S} = 128K$, $B_\mathrm{S} = 32$, $s = 4$, $h = 1024$, and LRU strategy.

We used the Intel compiler version 7.1 with switches:
`-O3 -fno_alias -pc64 -tpp6 -xK -ipo -align -Zp16`

Testing configuration 2

Some experiments were measured at Pentium Celeron M420 (Yonah) at 1.6 GHz, 1GB at 266 MHz, running OS Windows XP Professional with the following cache parameters:

- L1 data cache with $B_\mathrm{S} = 64$, $C_\mathrm{S} = 32K$, $s = 8$, $h = 64$, and LRU strategy.
- L2 unified cache with $B_\mathrm{S} = 64$, $C_\mathrm{S} = 1MB$, $s = 8$, $h = 2048$, and LRU strategy.

We used the Intel compiler version 9.0 with switches:
`-O3 -Og -Oa -Oy -Ot -Qpc64 -QxB -Qipo -Qsfalign16 -Zp16`

Testing configuration 3

Some experiments were measured at dual-core (only one thread=core was used) Intel i3-370M at 2.4 GHz, 4GB DDR3 at 532 MHz, running OS Windows 7 Home with the 32KB L1, 256KB L2, 3MB L3 cache. We used the GCC compiler version 4.5.3 with switches: `-O3`.

Testing configuration 4

Some experiments were measured at quad-core (only one thread=core was used) Intel i7-950 at 3.07 GHz, 24GB DDR3 at 1600 MHz, running OS Linux Ubuntu 12.04 with the 4*32KB L1, 4*256KB L2, 8MB L3 cache. We used the GCC compiler version 4.7.3 with switches: `-O3`.

Testing configuration 5

Some experiments were measured at a hypothetical processor with only L1 data cache with $C_{\mathrm{S}} = 1MB$. We tested this processor with the following values of the cache parameters:

5a) $B_{\mathrm{S}} = 32$, $s = 1$, $h = 32K$,
5b) $B_{\mathrm{S}} = 32$, $s = 4$, $h = 8K$,
5c) $B_{\mathrm{S}} = 1024$, $s = 4$, $h = 256$.

6. Results of an experimental evaluation

6.1. Performance evaluation of example codes

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

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

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

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

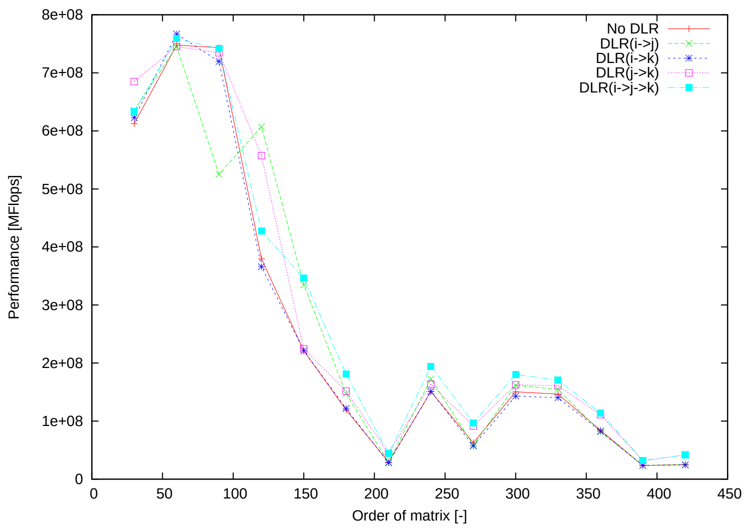

Fig. 5. Performance of `MMM_STD` for Testing configuration 1.

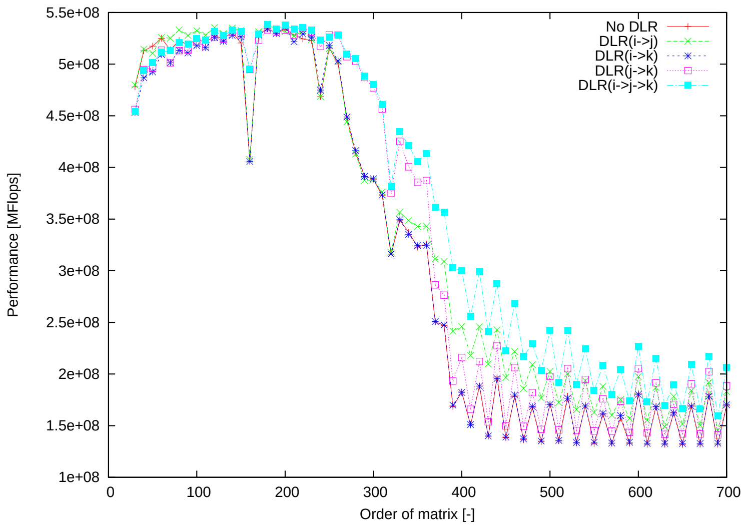

Fig. 6. Performance of `MMM_STD` for Testing configuration 2.

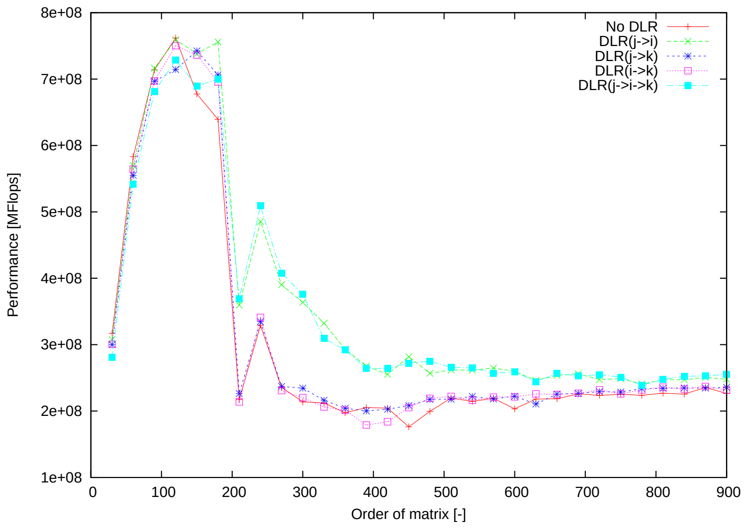

Fig. 7. Performance of `CHF_STD` for Testing configuration 1.

The graphs in Figures 5, 6, 7, and 8 illustrate the performance with or without DLR. These graphs illustrate that DLR increases the code performance due to better cache utilization. There is a performance gap (e.g. for $n = 210$ for the `CHF_STD` for Testing configuration 1), which can be overcome by DLR.

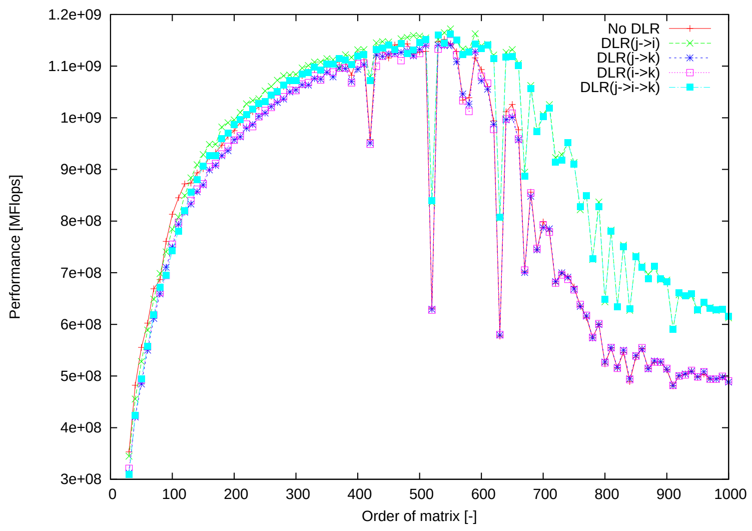

Fig. 8. Performance of `CHF_STD` for Testing configuration 2.

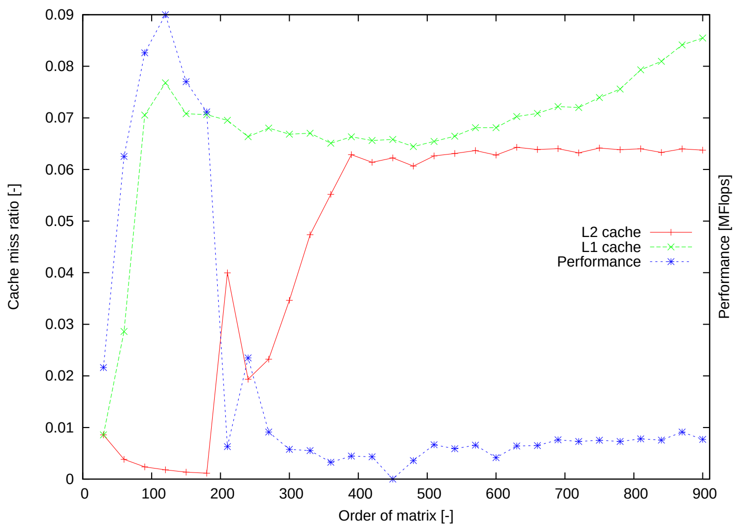

Fig. 9. Study of impact of cache utilization on performance of `CHF_STD` for Testing configuration 1.

The graphs in Figures 9, and 10 illustrate the impact of cache utilization on the performance of `CHF_STD` (the numbers of cache misses are measures by the software cache emulator [26]). We can conclude that the cache behavior has a great impact on the execution time, so the majority of spikes in the performace graph have the corresponding spikes in the cache utilization graph. Some spikes can not be interpreted in this way, they

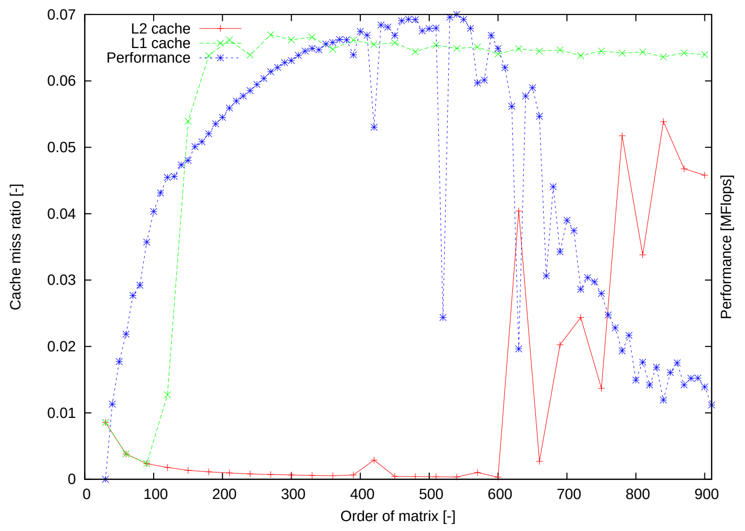

Fig. 10. Study of impact of cache utilization on performance of `CHF_STD` for Testing configuration 2.

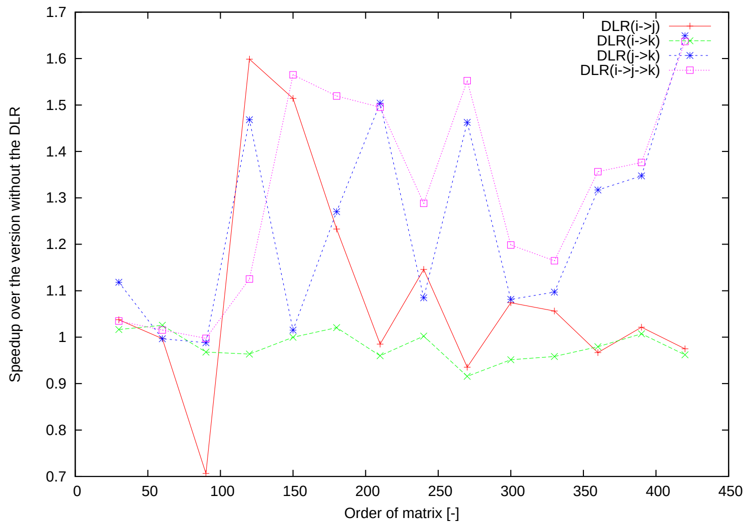

Fig. 11. Speedup of `MMM_STD` for Testing configuration 1.

arise due to the other reasons (translation lookaside buffer misses, unsuccessful compiler optimizations, etc.).

The graphs in Figures 11, 12, 13, 14, 15, and 16 show the speedup over the version without the DLR. We can conclude that the fastest code is the version with DLR($i \to j \to k$)

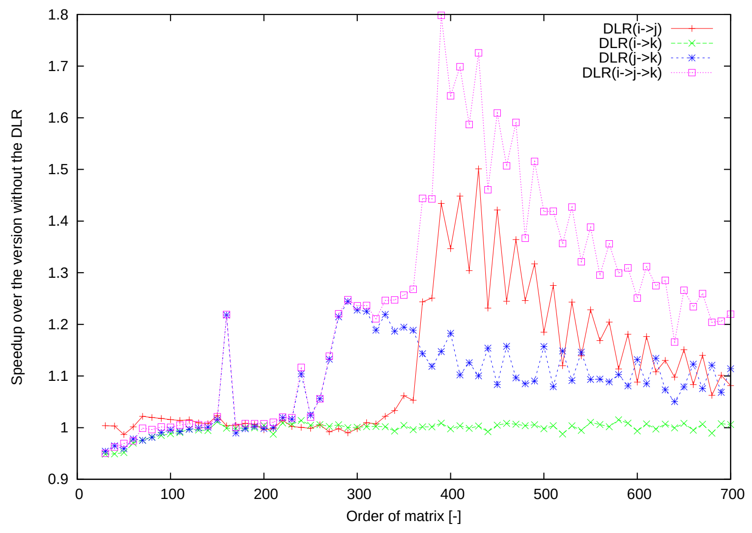

Fig. 12. Speedup of `MMM_STD` for Testing configuration 2.

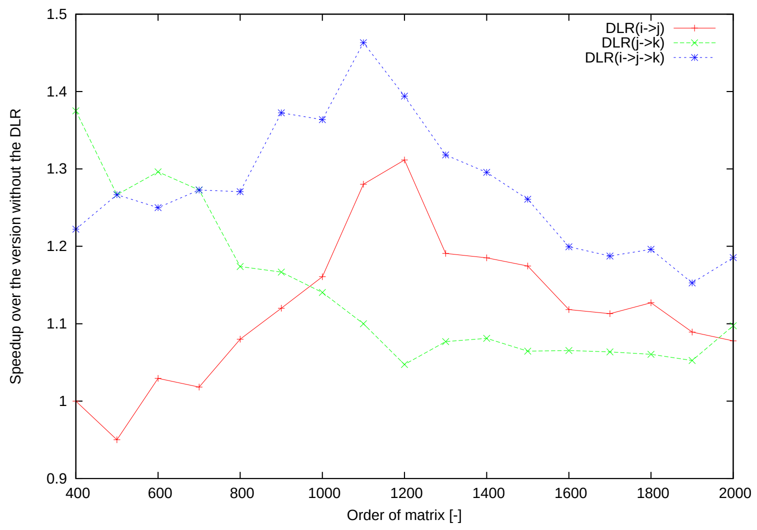

Fig. 13. Speedup of `MMM_STD` for Testing configuration 4.

for the `MMM_STD` code and with DLR($j \to i \to k$) for the `CHF_STD` code (for Testing configuraton 1 and 2), and with DLR($j \to i$) for the `CHF_STD` code (for Testing configuraton 4) . We can also conclude that the measured speedup is more than 20% in the measured set for the `MMM_STD` code and more than 10% for the `CHF_STD` code.

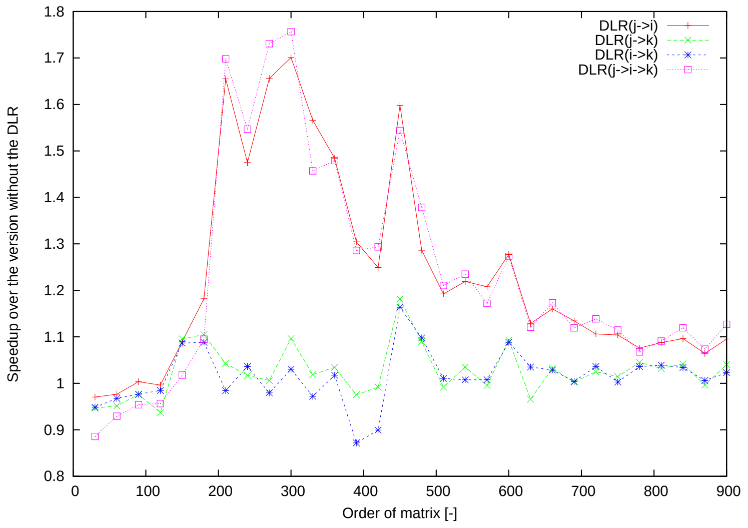

Fig. 14. Speedup of `CHF_STD` for Testing configuration 1.

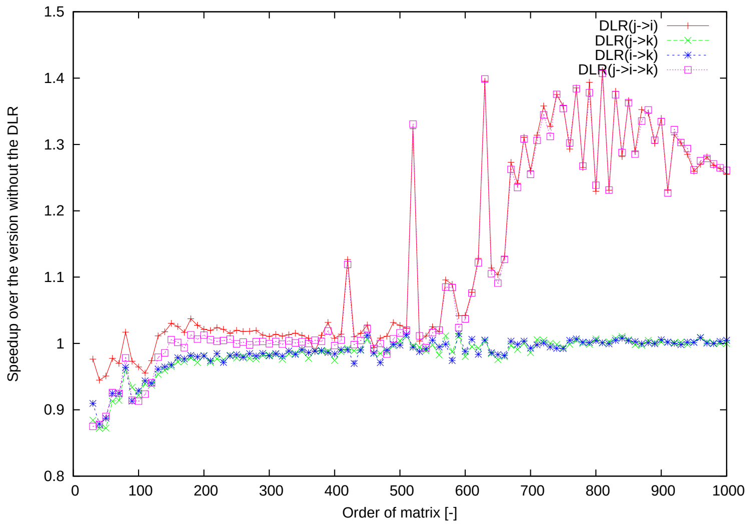

Fig. 15. Speedup of `CHF_STD` for Testing configuration 2.

The graphs in Figures 17 and 18 show the speedups over the version without DLR. We can conclude that DLR can accelerate `KP_STD`, `FW_STD`, and `MF_STD` codes.

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

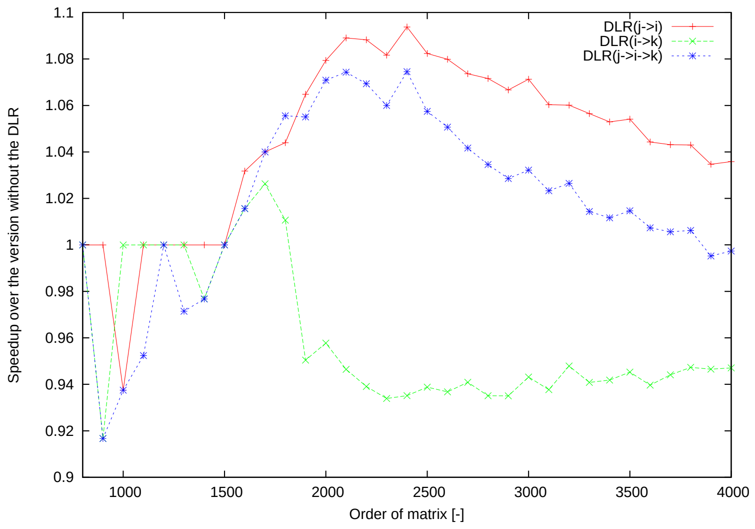

Fig. 16. Speedup of `CHF_STD` for Testing configuration 4.

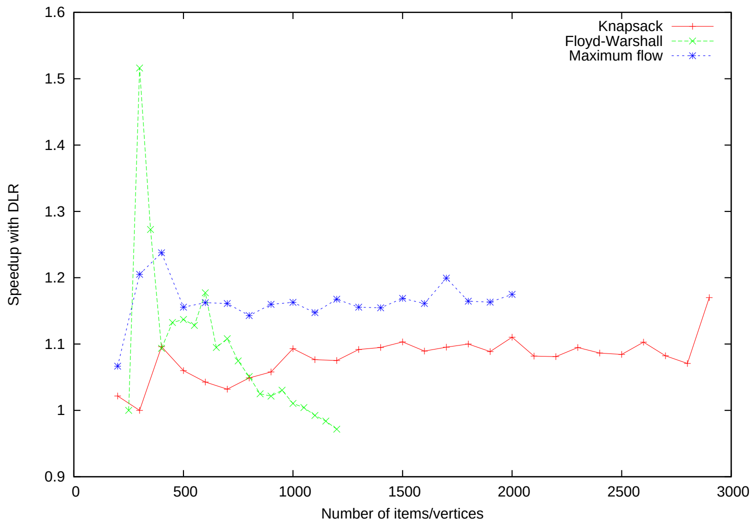

Fig. 17. Speedup of `KP_STD`, `FW_STD`, and `MF_STD` algorithms for Testing configuration 3.

6.2. Cache miss rate evaluation

The cache utilization is enumerated according to the following definitions.

$$\text{Relative number of cache misses} = \frac{\text{the number of cache misses with DLR}}{\text{the number of cache misses without DLR}}$$

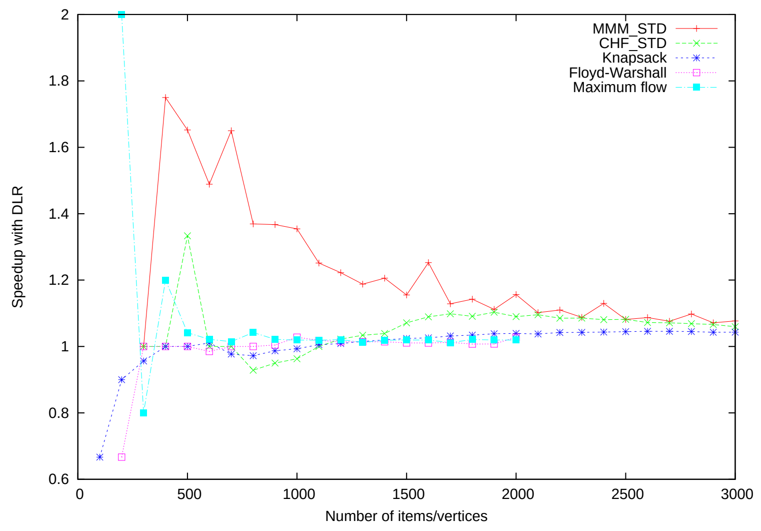

Fig. 18. Speedup of `MMM_STD`, `CHF_STD`, `KP_STD`, `FW_STD`, and `MF_STD` algorithms for Testing configuration 4.

The graphs on Figures 19, 20, 21, and 22 illustrate the number of cache misses occurring during one execution of the `MMM_STD` or `CHF_STD` pseudocodes. We can conclude that

– a 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 4.3)
– except for few cases, where the DLR transformation has a positive impact on cache utilization.

6.3. Evaluation of a simplified RD model based on GRD

Analytical cache model for `MMM_STD` To analyse this algorithm, we omit the accesses in an array C at a code line (6), 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 the type β and memory operations with $B[k][j]$ are of the type α.
– If DLR($j \to k$) is applied, then memory operations with $A[i][k]$ are of the type α and memory operations with $B[k][j]$ are of the type δ.

Analytical cache model for `CHF_STD` For analysing this algorithm, we omit the memory accesses outside the k-loop at a code line (8), because their contributions are negligible. Then, the algorithm contains the following types of memory accesses:

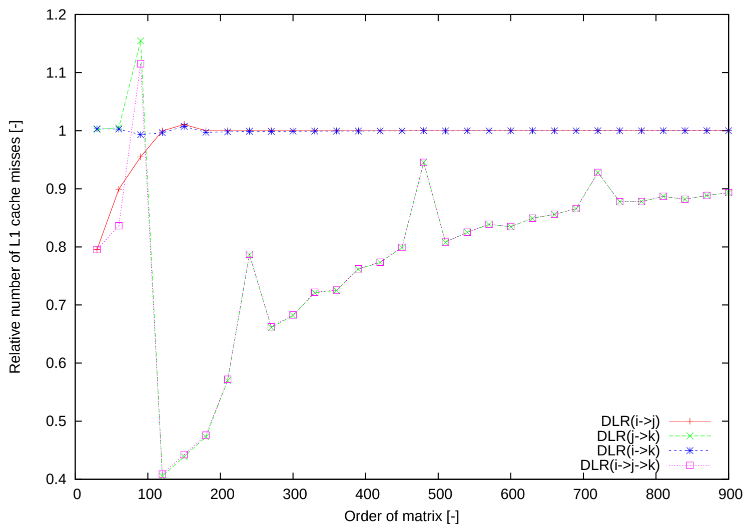

Fig. 19. Relative number of cache misses during MMM_STD for the L1 cache for Testing configuration 1.

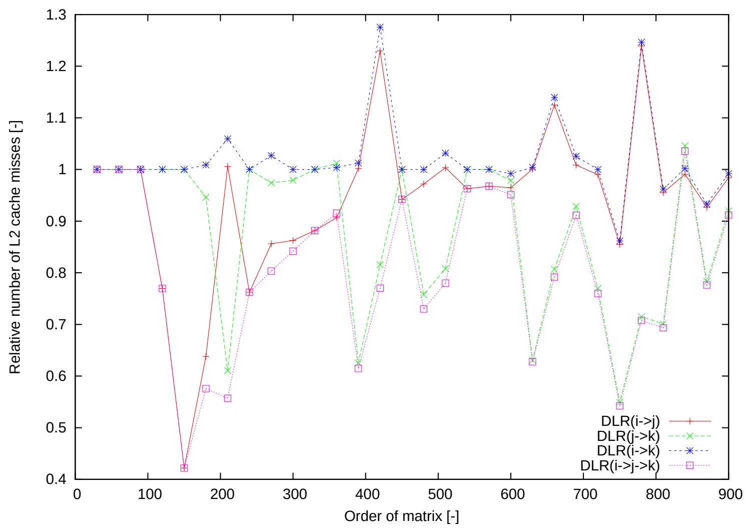

Fig. 20. Relative number of cache misses during MMM_STD for the L2 cache for Testing configuration 1.

- If DLR($j \to i$) is applied, then memory operations with $A[i][k]$ are of the type α and memory operations with $A[j][k]$ are of the type β.

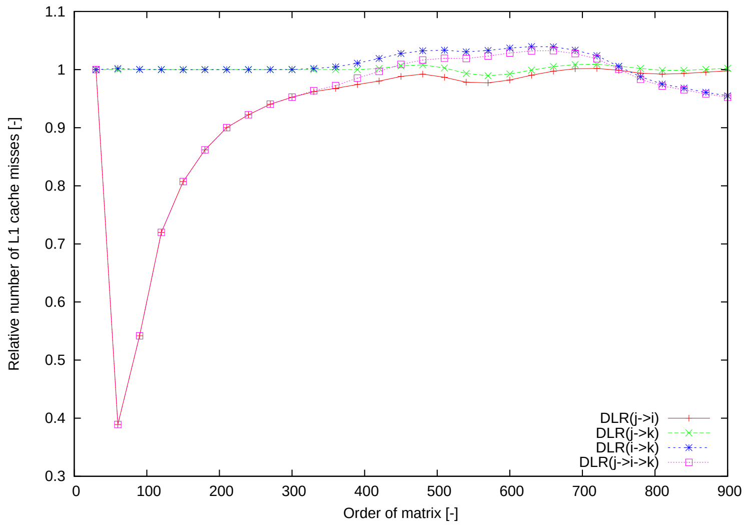

Fig. 21. Relative number of cache misses during `CHF_STD` for the L1 cache for Testing configuration 1.

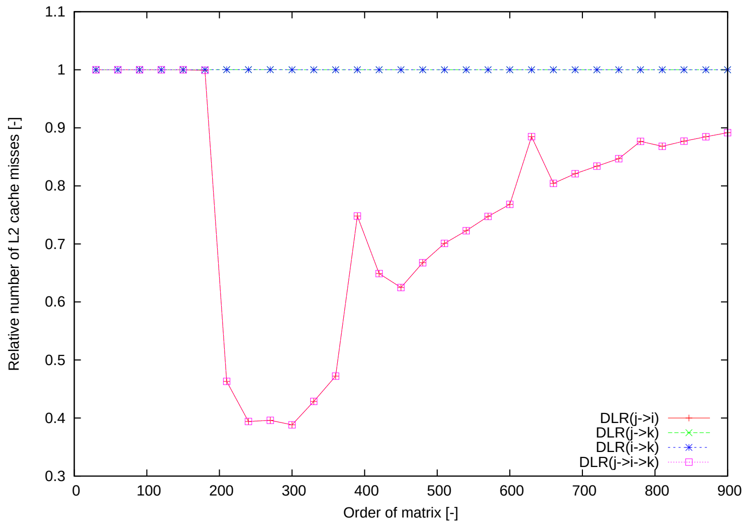

Fig. 22. Relative number of cache misses during `CHF_STD` for the L2 cache for Testing configuration 1.

– If DLR($k \rightarrow i$) is applied, then memory operations with $A[i][k]$ are of the type γ and memory operations with $A[j][k]$ are of the type α.

Analytical cache model for `spMMM_CSR`**,**`KP_STD`**, and** `MF_STD` The analysis of the cache behavior and DLR effects for these algorithms are beyond the scope of the compiler due to its irregular memory pattern.

Example of the 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 the type α and memory operations with $B[k][j]$ are of the type δ.

Firstly, we must count how many iterations of the j-loop can reside in the cache. From the types of memory operations (Eq. (2) on Page 1392), 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 the cache parameters in Eq. (1) on Page 1392)

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

The number of cache misses saved by DLR(k, j) per one iteration of the j-loop (Eq. (4) on Page 1392) 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 Testing configuration 1:
  - for L1 cache: $N_{\mathrm{iter}} = 228$.
  - for L2 cache: $N_{\mathrm{iter}} = 3640$.
- for Testing configuration 2:
  - for L1 cache: $N_{\mathrm{iter}} = 455$.
  - for L2 cache: $N_{\mathrm{iter}} = 14564$.

The discussion on the precision of the simplified RD model Comparisons of the numbers of estimated and measured cache misses are shown in Figures 23, 24, 25, 26, and 27. Our analytical model is derived from GRD which is based on a fully-associative cache memory assumption. This assumption is the main source of errors in predictions. The errors are higher for the caches with low associativity (e.g. the direct-mapped cache in Testing configuration 5a, Figure 25). On the other hand, our model is relatively precise (e.g. the caches in Testing configuration 5b or 5c, Figures 26 and 27) for caches with higher associativity (i.e., $s \geq 4$). This condition is true for the majority of modern CPU caches.

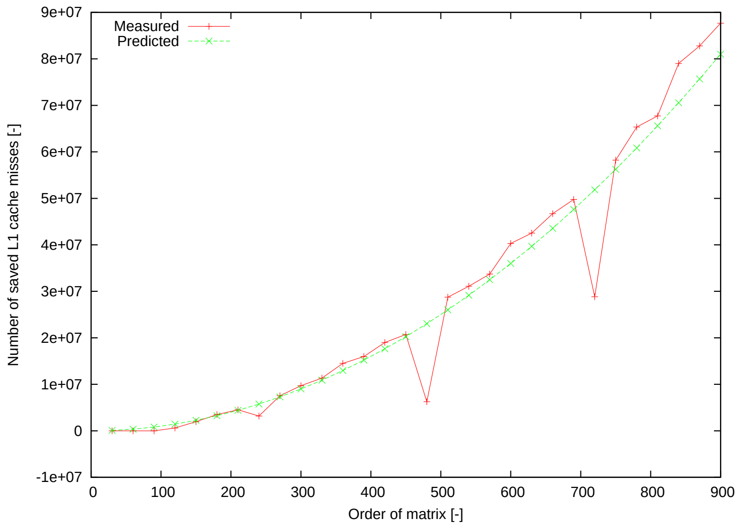

Fig. 23. Comparison of the numbers of estimated and measured cache misses (μ_{saved}) saved by DLR during the execution of `MMM_STD` for the L1 cache.

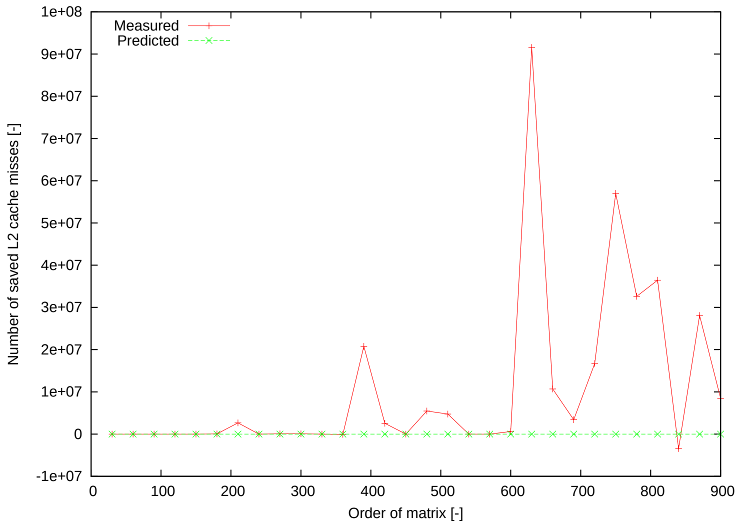

Fig. 24. Comparison of the numbers of estimated and measured cache misses (μ_{saved}) saved by DLR during the execution of `MMM_STD` for the L2 cache.

6.4. Evaluation of the combination of DLR and loop tiling

We have also measured the performance and cache utilization for the pseudocode `MMM_TIL` and the effects of DLR transformation on this code.

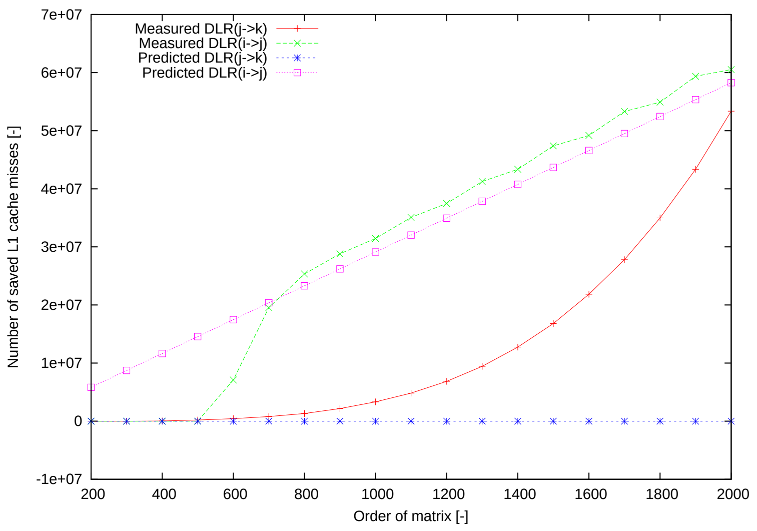

Fig. 25. Comparison of the numbers of estimated and measured cache misses (μ_{saved}) saved by DLR during the execution of `MMM_STD` for Testing configuration 5a).

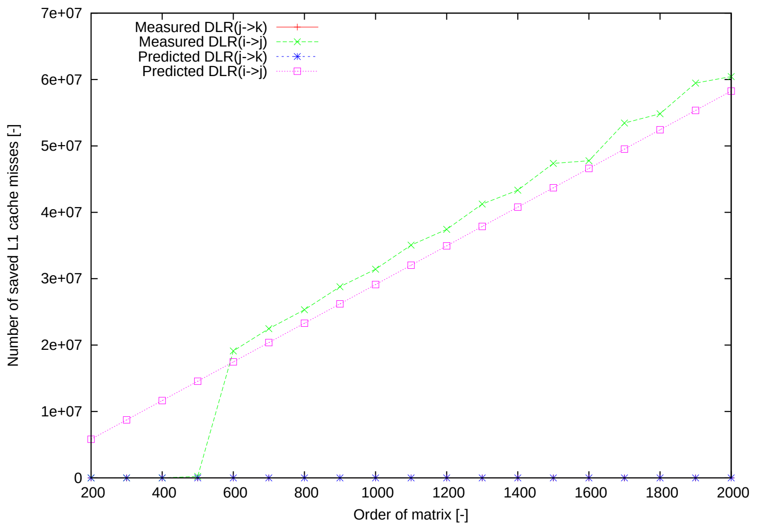

Fig. 26. Comparison of the numbers of estimated and measured cache misses (μ_{saved}) saved by DLR during the execution of `MMM_STD` for Testing configuration 5b).

Graphs in Figures 28 and 29 illustrate the fact that the 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

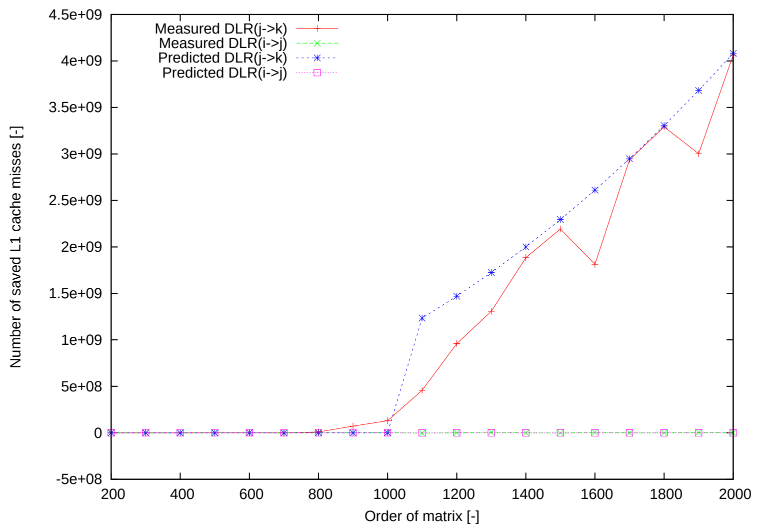

Fig. 27. Comparison of the numbers of estimated and measured cache misses (μ_{saved}) saved by DLR during the execution of `MMM_STD` for Testing configuration 5c).

from the optimal value. When 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 the cases when it is hard to predict a good value for the tiling factor.

6.5. Evaluation of DLR for the `spMMM_CSR` code

The `spMMM_CSR` code is a simple example of an irregular code. In this code, the memory access pattern is hard to predict on the compiler level and loop tiling is excluded. But DLR is usable and the application of this technique can save a reasonably large number of the cache misses (see Figures 30, 31, and 32). The numbers of L1 cache misses remain the same (without and with DLR), so graphs are shown only for the L2 cache utilization.

7. Automatic compiler support of DLR

The DLR transformation brings new possibilities to optimize the nested loops.

7.1. Proposed algorithm of an automatic compiler support of DLR

We assume that the optimizing compiler evaluates each loop nest (for details see [33]) independently. Let $\mathcal{L}_{1...b}$ represent a b-dimensional loop nest (hierarchy of immediately nested loops), where $\mathcal{L}_1$ is the outermost loop and $\mathcal{L}_b$ is the innermost loop. The control variable for the loop $\mathcal{L}_i$ is denoted as $\mathcal{C}_i$. We propose the following function that returns a list of the loop numbers that can profit from the DLR application. We plan to implement more specific variant of this function into the compiler to support the DLR application automatically.

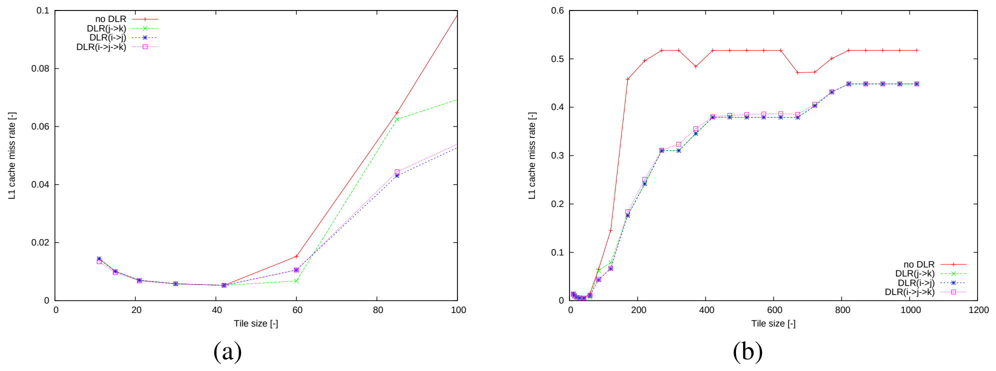

Fig. 28. The L1 miss rate for an `MMM_TIL` algorithm (for n=1024) for small and large values of the tile size.

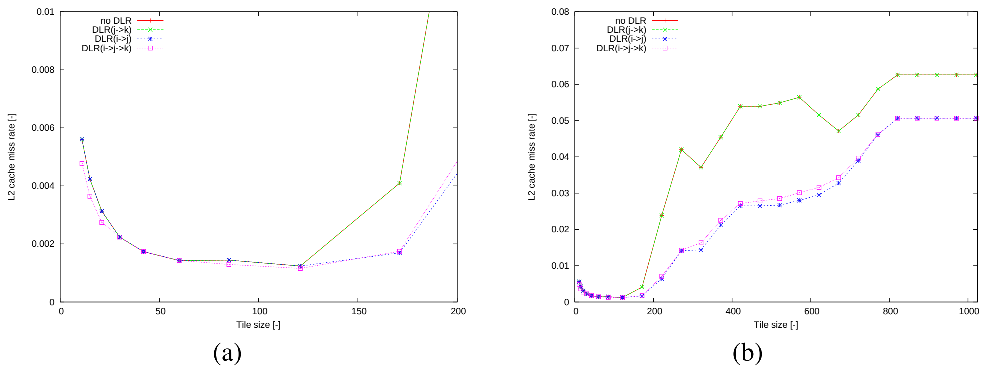

Fig. 29. The L2 miss rate for an `MMM_TIL` algorithm (for n=1024) for small and large values of the tile size.

```
1: procedure DLR_APPLICATION(in $b, \mathcal{L}, \mathcal{C}$;out $res$)
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          ▷ i.e., loop $\mathcal{L}_{i+1}$ is reversible
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 $res$;
```

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

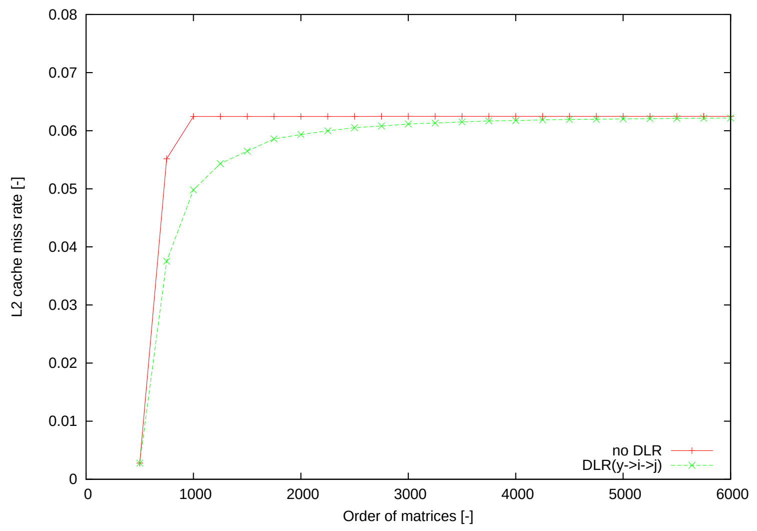

Fig. 30. The L2 cache miss rate for an `spMMM_CSR` algorithm (for $density(A) = 7\%$ and $density(B) = 21\%$)

the code performance. In real applications, the benefits of DLR application should be compared to the other source code optimizations.

7.2. Discussion on the DLR applicability inside the compilers

The function in Section 7.1 is very brief. The compiler optimization must address more issues:

- Where can DLR be applied? DLR can be applied on all nested reversible loops (excluding outermost loops). This condition can be easily checked by the compiler.
- Where should DLR be applied? DLR should be applied on a pair or triple of loops, which causes a maximal effect on cache behavior (mentioned in Section 3.3). This compiler decision is very similar as the one for the loop tiling.
- Does DLR have a significant effect? Yes. In most cases, higher speedups are achieved by the loop unrolling or loop tiling. But DLR can be combined with these techniques (see Section 6.4), and moreover, it can be applied on the codes, where the application of the loop tiling is not possible (see Section 6.5).
- Is DLR supported in any compiler system? No, providing an automatic support of DLR is a very difficult task. We have tried to develop the DLR support as a new module for a LLVM project. The GCC system is the most popular compiler, but there are strong connections between compilation stages, so add the support for other source code optimization is very complicated. We also consider the DLR support in other source transformation systems (e.g. Rascal Metaprogramming Language [16]). During the (ongoing) implementation occur the following problems:

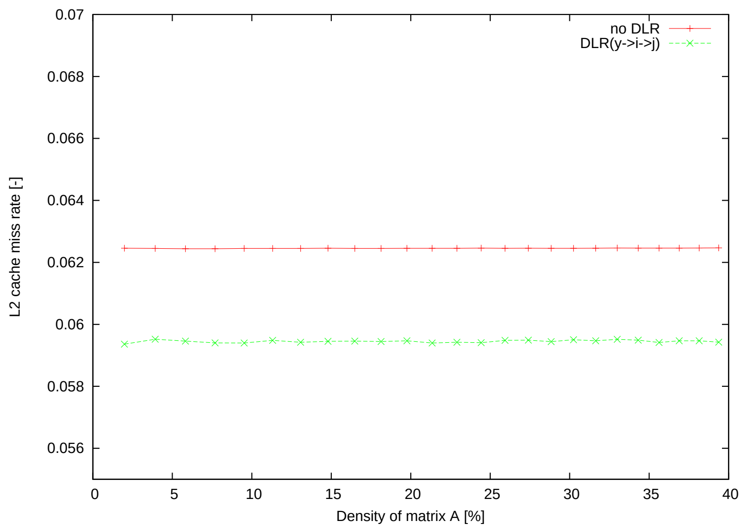

Fig. 31. The L2 cache hit rate for an algorithm `spMMM` (for n=2200 and $density(B) = 17\%$)

- The DLR pays off for the majority of measured situations, but the exact threshold depends strongly on the measured problem and also on the exact configuration for the measurement. The model of the cache behavior is introduced, but it can predict the number of “saved”cache misses only for some types of accesses (this model is unusable e.g. for the indirect access etc.)
- The presented model can predict “saved”cache misses, but the DLR transformation also increases the number of conditional branches. Furthermore, effects of these cache misses or conditional branches are influenced by the other compiler optimizations.
- The explicit use of DLR can confuse optimizing compilers and hinder further loop optimizations (since it “destroys”perfectly nested loops etc.). So, it should be applied as the last possible high-level optimization.

8. Conclusions

We have described a new code transformation technique, the dynamic loop reversal, whose goal is to improve the locality of the accessed data and improve the cache utilization. This transformation seems to be very useful for the codes with nested loops. We have demonstrated significant performance gains for six basic algorithms from linear algebra, graph theory etc.

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 occur mainly due to the fully-associative cache memory assumption.

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

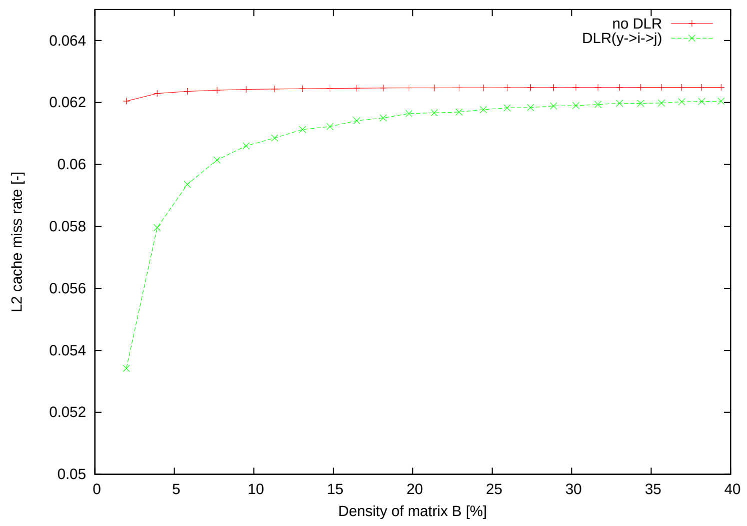

Fig. 32. The L2 cache hit rate for an algorithm `spMM` (for n=3700 and $density(A) = 10\%$)

Acknowledgments. This work was supported by RVO|18000|ext:|FIS:110 − 111000| grant of MŠMT.

References

1. Ahmed, N., Mateev, N., Pingali, K.: Tiling imperfectly-nested loop nests. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM). SC '00, IEEE Computer Society, Washington, DC, USA (2000), `http://dl.acm.org/citation.cfm?id=370049.370401`
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK user's guide (1999), `http://epubs.siam.org/doi/abs/10.1137/1.9780898719604`
3. Beyls, K.: Exact compile-time calculation of data cache behavior. `http://www.elis.rug.ac.be/aces/edegem2002/beyls.pdf`
4. Beyls, K., D'Hollander, E.: Reuse distance as a metric for cache behavior. In: Proceedings of PDCS'01. pp. 617–662 (August 2001), `citeseer.ist.psu.edu/beyls01reuse.html`
5. Carr, S., Kennedy, K.: Blocking linear algebra codes for memory hierarchies. In: Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing. pp. 400–405. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1990), `http://dl.acm.org/citation.cfm?id=645819.669407`
6. Carr, S., Lehoucq, R.: Compiler blockability of dense matrix factorizations. ACM Transactions on Mathematical Software 23, 336–361 (1996)
7. Carr, S., McKinley, K.S., Tseng, C.W.: Compiler optimizations for improving data locality. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 252–262 (1994)

8. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. pp. 4:1–4:12. SC '08, IEEE Press, Piscataway, NJ, USA (2008), http://dl.acm.org/citation.cfm?id=1413370.1413375
9. Dongarra, J.J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software 16(1), 1–17 (Mar 1990)
10. Golub, G.H., Van Loan, C.F.: Matrix Computations (3rd ed.). Baltimore: Johns Hopkins (1996)
11. Im, E.: Optimizing the Performance of Sparse Matrix-Vector Multiplication - dissertation thesis. University of California at Berkeley (2001)
12. Šimeček, I., Tvrdík, P.: Sparse matrix-vector multiplication — final solution? In: Parallel Processing and Applied Mathematics. PPAM'07, vol. 4967, pp. 156–165. Springer-Verlag, Berlin, Heidelberg (2008), http://www.springerlink.com/content/48x1345471067304/
13. Šimeček, I., Tvrdík, P.: Dynamic loop reversal — the new code transformation technique. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 1587–1594 (2013)
14. John L. Hennessy, D.A.P.: Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann; 4 edition (September 27, 2006) (2006), http://www.amazon.com/Computer-Architecture-Fourth-Quantitative-Approach/dp/0123704901/ref=pd_sim_b_1
15. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc. (2002)
16. Klint, P., van der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on. pp. 168–177 (Sept 2009)
17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005), http://doi.acm.org/10.1145/1118890.1118892
18. Press, H.W., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing. Cambridge University Press (2007)
19. Seward, J., Nethercote, N., Hughes, T., Fitzhardinge, J., Weidendorfer, J., Mackerras, P., et al.: Valgrind documentation, http://valgrind.org/docs/manual/valgrind_manual.pdf
20. Song, Y., Li, Z.: New tiling techniques to improve cache temporal locality. SIGPLAN Not. 34, 215–228 (May 1999), http://doi.acm.org/10.1145/301631.301668
21. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The pochoir stencil compiler. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 117–128. SPAA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1989493.1989508
22. Tuma, M.: Overview of direct methods. I. Winter School of SEMINAR ON NUMERICAL ANALYSIS (January 2004)
23. Tvrdík, P., Šimeček, I.: Analytical modeling of optimized sparse linear code. In: Parallel Processing and Applied Mathematics. vol. 3019/2004, pp. 207–216. Czestochova, Poland (2003), http://www.springerlink.com/content/drwdhen7db199k05/
24. Tvrdík, P., Šimeček, I.: 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, pp. 190–197. Montreal, Canada (2004), http://dl.acm.org/citation.cfm?id=1018426.1020360
25. Tvrdík, P., Šimeček, I.: A new diagonal blocking format and model of cache behavior for sparse matrices. In: Proceedings of the 6th International Conference on Parallel Processing

and Applied Mathematics. PPAM'05, vol. 12, pp. 164–171. Springer-Verlag, Poznan, Poland (2005), `http://dl.acm.org/citation.cfm?id=2096870.2096894`
26. Tvrdík, P., Šimeček, I.: Software cache analyzer. In: Proceedings of Czech Technical University Workshop. vol. 9, pp. 180–181. Prague, Czech Republic (mar 2005)
27. Vera, X., Xue, J.: Efficient compile-time analysis of cache behaviour for programs with IF statements. In: International Conference on Algorithms And Architectures for Parallel Processing. pp. 396–407. Beijing (October 2002), `citeseer.ist.psu.edu/567600.html`
28. Vera, X., Xue, J.: Let's study whole-program cache behaviour analytically. In: Proceedings of the 8th International Symposium on High-Performance Computer Architecture. pp. 175–186. HPCA '02, IEEE Computer Society, Washington, DC, USA (Feb 2002), `http://dl.acm.org/citation.cfm?id=874076.876456`
29. Vuduc, R., Demmel, J.W., Yelick, K.A., Kamil, S., Nishtala, R., Lee, B.: Performance optimizations and bounds for sparse matrix-vector multiply. In: Proceedings of Supercomputing 2002. Baltimore, MD, USA (November 2002)
30. Wadleigh, K.R., Crawford, I.L.: Software optimization for high performance computing. Hewlett-Packard professional books (2000)
31. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. SIGPLAN Not. 26, 30–44 (May 1991), `http://doi.acm.org/10.1145/113446.113449`
32. Wolfe, M.: High-Performance Compilers for Parallel Computing. Addison-Wesley, Reading, Massachusetts, USA (1995)
33. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers, Norwell, MA, USA (2000)

Ivan Šimeček 2009-now Assistant Professor at the Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague (CTU in Prague).
2009-now Executive manager of the Prague Nvidia CUDA teaching center.
* Author of more than 40 scientific journals and conference publications.
* Main research areas: GPGPU, distributed algorithms, sparse matrix storage formats, numerical linear algebra, cache memory models, programming languages, applied crystallography.

Pavel Tvrdík 2009–now professor, Department of Computer Systems, FIT CTU in Prague.
2009–now dean, FIT CTU in Prague.
2009-now member of scientific councils of FIT CTU in Prague
* Author or co-author of 10 papers in journals in WoS and over 50 papers in proceedings of refereed conference proceeding. Co-editor of 1 book in IEEE Computer Science Press. Documented over 65 citations in WoS.
* A member of editorial board of Scalable Computing: Practice and Experience (since 2005).
* Member of IPCs of more than 50 international conferences, including: ICCS'11, PDCN'11, PDCS'11, HPCS'10, LaSCoG-SCoDiS'10, PDCN'10, ICCS'10, PPAM'09, PDCS'09, PDCN'09, ICCS'09, PDCS'08, PDCN'08, ICCS'08, PDCN'07, ICCS'07, PPAM'07, PDCS'07, HPCS'07, ICCSA'07, PDCS'06, PDCN'06, InfoScale'06, ICCS'06, ICCSA'06.
* Main research areas: Parallel architectures and algorithms, programming languages.

Received: November 26, 2013; Accepted: September 1, 2014.