

Multilevel Blocking in Complex Iteration Spaces

M. Jiménez, J. M. Llabería, A. Fernández and E. Morancho

Departamento de Arquitectura de Computadores, Universidad Politécnica de Cataluña
Gran Capitán s/n, Módulo D6, E-08034 Barcelona, (Spain), e-mail: marta@ac.upc.es

Abstract

This paper presents a new unified method for simultaneously tiling the register and cache levels of the memory hierarchy. We will focus on the code transformation phase of tiling. Our algorithm uses strip-mining and loop interchange on all memory hierarchy levels to determine the tiles as usual, and, afterwards, and due to the special characteristics of the register level, we apply index set splitting, fully unrolling and unnecessary load/store elimination.

We propose a technique to perform the loop interchange in non-convex iteration spaces that computes the loop bounds exactly and we also present an order in which to perform index set splitting that guaranties that each loop in the nest will be processed only once and also avoids code explosion.

The performance of the memory hierarchy obtained with the method presented in this paper is substantially higher than the performance obtained by commercial preprocessors capable of restructuring code to exploit the memory hierarchy.

1 Introduction

To reduce the average memory access time, today's and future computer architectures use several memory levels. The memory size in each level of the hierarchy and the program locality determine the effectiveness of the memory hierarchy[12]. Since the number of memory levels is continuously increasing, it is important to exploit as much as possible every level of the memory hierarchy.

Numerical codes with big data structures are sensitive to the performance of the memory hierarchy since their data structures don't fit in the memory levels near to the processor. Nevertheless, numerical codes usually have spatial and temporal locality properties. Block and multilevel block algorithms have been studied to increase data reuse and to improve the performance of the memory hierarchy [5][15][14].

In order to use computer architectures with several memory levels, it is important to hide the details of the architecture from the user, making the memory hierarchy transparent to the user. Therefore several code transformation techniques have been developed. These techniques aim at exploiting the temporal and spatial locality properties of a program. Spatial locality can be exploited accessing array elements in the same order that they are stored in memory. Temporal locality can be exploited using loop tiling.

Iteration space tiling is a code restructuring technique used to reduce a program's data working set. It consists in dividing the iteration space defined by the loop structures in regular tiles. Tiling the iteration space creates a blocking of the data arrays. The order in which the tiles are traversed induces the order in which the data blocks are accessed. The idea is to shorten the distance between successive references to the same memory location, so that it is more probable that the memory word resides in the memory levels near to the processor when the data reuse occurs. To determine the tiling (tile size and tile order) in each level, a locality analysis must be performed [3][9][18].

Tiling an iteration space can be implemented using unroll & jam and scalar replacement at the register level [2][4], and applying strip-mining and loop interchange at all other levels of the hierarchy [5][6][18]. Nevertheless, tiling complex iteration spaces presents several problems. Applying strip-mining to a loop nest always produces a non-convex iteration space since some of the loops in the nest will end up with a step size different from 1. Therefore, the loop interchange needed after strip-mining can not be done using techniques based on unimodular transformation matrices [19].

Previous work on tiling, such as Wolf and Lam [19] and M. Wolfe[20], uses ad-hoc methods to implement the loop interchange after strip-mining, and may create empty tiles when the iteration space is not rectangular. Wolf and Lam compute the bounds of the tile loops finding the minimal rectangle that contains the original iteration space. Using this method, some of the tiles may be empty. Wolfe computes the bounds of the tile loops exactly, therefore, there is no empty tile, but his technique may iterate over empty points inside the tile. The time wasted in executing those empty loops can be noticeable if tiling is performed in more than one level of the memory hierarchy. The effect of not computing the loop bounds exactly is multiplicative with respect to the number of levels exploited.

On the other hand, the ad-hoc methods used by them to compute the loop bounds after the loop interchange only work if the relative order of the loops after the interchange is the same as in the loop nest before strip-mining. Therefore, in general, they need to perform the loop interchange transformation so many times as memory levels they are exploiting.

At the register level, S. Carr [4] uses pattern recognition techniques on the bounds of the loops to apply unroll & jam. Nevertheless, when the iteration space is not rectangular, the bounds of the loops can not be matched by the patterns and no general algorithm to partition these iteration spaces into simpler ones that could be recognized through patterns is presented in [4]. We call complex iteration spaces the iteration spaces that are not rectangular.

Commercial preprocessors such as KAP from Kuck & Associates are not always able to produce an iteration space tiling when the bounds of the loops are affine functions (or compositions of affine functions) of the surrounding loops iteration variables. These types of bounds are commonly found in linear algebra algorithms or arise as a result of applying transformations such as loop skewing.

This paper presents a new unified method for simultaneously tiling the register and cache levels of the memory hierarchy, that works for any iteration space having loop bounds defined as compositions of affine functions of the surrounding loops iteration variables (complex iteration spaces). We will focus on the code transformation phase of tiling and we will assume that dependency analysis and locality analysis have already been performed [9][18]. Strip-mining and interchange is performed in a single step for all levels of the memory hierarchy simultaneously and the loop bounds are computed exactly. Also, we use index set splitting instead of pattern recognition techniques at the register level.

In section 2 we present the framework where we develop our code restructuring technique. The transformation steps of the unified process are presented in section 3. In section 4 we present the evaluation of the restructuring process and in section 5 we expose the conclusions.

2 Framework

We consider algorithms specified by a number of nested loops. The set of iterations determined by the bounds of the n nested loops is a convex subset of Z^n . We call this set the Bounded Iteration Space (BIS): $BIS = \{I=(i_1, \dots, i_n)^t \mid L_1 \leq i_1 \leq U_1, \dots, L_n \leq i_n \leq U_n\}$

where I is a n -dimensional vector which represents any single iteration of the n loops and $L_i(U_i)$ is the lower (upper) bound of loop i . The bounds of the loops are max or min functions of affine functions of the surrounding loops iteration variables.

The BIS can be specified in a matrix form [10] as follows: $A \cdot I \leq \beta$, where every row of matrix A defines an affine function of the lower or the upper bound. The n elements of vector I are the iteration control variables i_k , and β is a vector whose components are the independent terms of each affine function.

A transformation, represented by matrix T , maps each iteration I of BIS to one iteration J of the Bounded Transformed Iteration Space (BTIS): $BTIS = \{ J = T \cdot I \mid I \in BIS \}$

T is a valid transformation if T is non singular and the dependences are preserved [7].

The Minimum Convex Space (MCS) which contains all the points of the BTIS can be put in matrix form, using the transformation matrix T and the matrix inequation which represents the bounds of the BIS:

$$\begin{array}{lcl} A \cdot I \leq \beta & \Rightarrow & A \cdot T^{-1} \cdot J \leq \beta \\ T \cdot I = J & & A' \cdot J \leq \beta \end{array}$$

The bounds of the MCS can be extracted from the matrix inequation $A' \cdot J \leq \beta$ using the Fourier-Motzkin algorithm [1].

When T is unimodular, the bounds of MCS are the bounds of the BTIS and can be directly used to build the loop nest required to scan the BTIS (we say that these bounds are exact). When T is non-unimodular there are holes (integer points without integer antiimage) in the BTIS. To scan correctly the BTIS, all the holes must be skipped. The bounds of the MCS must be corrected to obtain the exact bounds of the BTIS, required to build the transformed loop nest. In [7] a method to compute the exact bounds of the transformed loop nest is presented.

3 Transformation Steps

In this paper we will focus on the code transformation phase of tiling and we will assume that locality and reuse analysis to define the tiles (the form and the size) in each level of the memory hierarchy have already been performed [13][9][18]. We also assume that the loops to be transformed are fully permutable or a transformation has been previously used to achieve it [17].

The steps we follow are:

1. Create a perfectly nested loop.

Tiling: To all levels of the memory hierarchy (cache and registers)

2. Apply strip-mining to all loops selected by the locality and reuse analysis.
3. Interchange loops as determined by the locality and reuse analysis.

At the register level:

4. Apply index set splitting repeatedly until being able to fully unroll those loops that provide data reuse at the register level.
5. Distribute the surrounding loop of the loops we want to fully unroll [20].
6. Remove all redundant loop bounds and empty loops.
7. Fully unroll all innermost loops.
8. Eliminate unnecessary loads/stores using scalar replacement [4].

When the original loop nest is not perfectly nested, we use a code sinking transformation [20]. This transformation moves the statements between loops inside the inner loop by adding one or more conditional statements. The conditional statements protect the execution of the merged statements, so they are executed at the right time. To avoid the time execution overhead due to these conditionals, after tiling the loop nest, they are eliminated using index set splitting [11].

Strip-mining determines the tile size in each dimension and interchange determines the order in which the tiles are traversed to increase the data reuse between tiles. After interchange, the innermost loops provide the data reuse at the register level, and the outermost loops provide the data reuse in the last considered memory level of the memory hierarchy.

After tiling the iteration space, we want to exploit the data locality at the register level. Due to the special characteristic of the register level, it is necessary to fully unroll the loops that provide data reuse in this level and to eliminate unnecessary loads/stores in the innermost loop. Index set splitting [20] is used to divide the loops in zones in such a way that in one zone all the loops that provide data reuse at the register level can be fully unrolled. A loop is fully unrolled replicating the loop body as many times as the bounds indicate.

Index set splitting can increase the complexity of the bounds of the loops and can create empty loops unnecessarily. To remove empty loops and redundant loop bounds, we apply the Fourier-Motzkin algorithm to each loop nest of the transformed code [11].

At last, scalar replacement [2] is used to eliminate unnecessary loads/stores in the innermost loop. Scalar replacement finds opportunities for reuse of subscripted variables and replaces the references involved by array references to temporal scalar variables; among other cases, we use scalar replacement for array elements that are loop invariant with respect to the iteration variable of the innermost loop. Sometimes it is necessary to distribute the surrounding loops to apply scalar replacement.

Strip-mining, index set splitting and unrolling can always be performed because execution order is unchanged. Loop interchange, loop distribution and scalar replacement are always legal since we assume that the loops are fully permutable.

3.1 Tiling: Strip-mining and interchange

Strip-mining decomposes a loop into two nested loops; the outer loop steps between strips of consecutive iterations, and the inner loop traverses the iterations within a strip [20]. We will refer to the outer loop as "the tile loop" (TI loop) and to the inner loop as the "the element loop" (EL loop). After strip-mining, the loops are permuted in such a way that the TI loops become the most external loops and the EL loops the most internal ones.

If strip-mining is applied in a straightforward manner to a complex iteration space (i. e. triangular) the shape of the resulting tiles might not be as desired [21]. We will apply the strip-mining transformation as defined by the following expression:

$$\begin{aligned} \text{do } i = L_i, U_i & \xrightarrow{\text{Strip-Mining}} \text{do } i^B = \lfloor (L_i - \text{oft}_i \bmod B_i) / B_i \rfloor \cdot B_i + \text{oft}_i \bmod B_i, U_i, B_i \\ & \text{do } i = \max(i^B, L_i), \min(i^B + B_i - 1, U_i) \end{aligned}$$

where i^B is the TI loop, i is the EL loop, B_i is the strip size and $\text{oft}_i \in \mathbb{Z}$ is an offset. Using this expression, the tile boundaries are parallel to the iteration space axes and the offset determines the origin of the first tile. For simplicity and without loss of generality we will assume during the explanation a null offset.

To better clarify the steps performed in this section we will use the code of Fig. 1(a) as an example. After applying strip-mining to loops i and j , we obtain the code of Fig. 1(b). We will

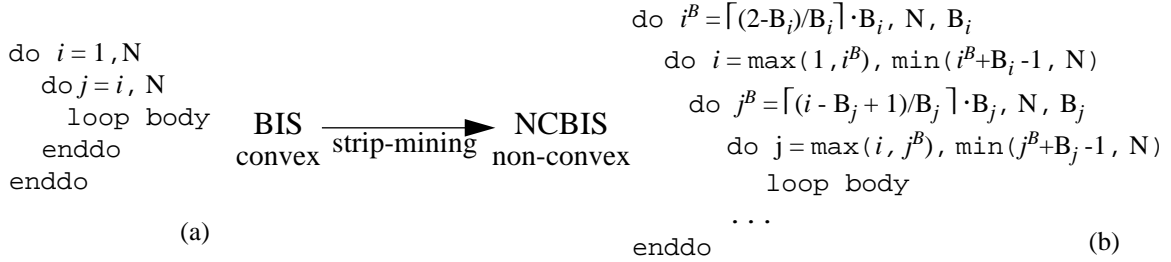


Fig. 1. (a) Example of loop nest (BIS). (b) Loop nest after applying strip-mining to loops i and j (NCBIS). refer as NCBIS (Non-Convex Bounded Iteration Space) to the non-convex iteration space resulting from applying the strip-mining transformation to BIS.

Loop interchange orders the loops achieving high data reuse between tiles. We would like to obtain the BTIS (Bounded Transformed Iteration Space) which is the iteration space after applying the interchange transformation to NCBIS.

The iteration space after strip-mining is a non-convex space and, therefore, the interchange unimodular matrix (T_I) cannot be directly applied using the theory of unimodular transformations [19]. To perform the interchange transformation we will use the theory of non-unimodular transformations described in [7] which is able to deal with non-convex iteration spaces.

We suppose NCBIS comes from a certain original convex space, that we will refer to as OBIS (Original Bounded Iteration Space), after applying a non-unimodular transformation (T_a) to it (Fig. 2). To obtain the BTIS, a non-unimodular transformation T is applied to OBIS. This non-unimodular transformation is a composition of the interchange unimodular transformation T_I and the non-unimodular transformation T_a ; that is $T = T_I \cdot T_a$. In Fig. 2 a diagram of the transformation process is shown.

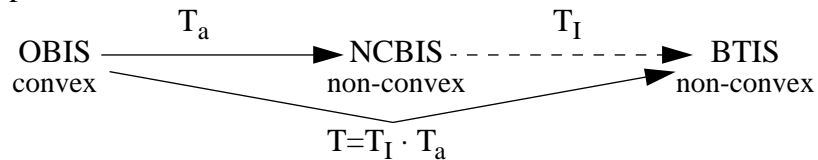


Fig. 2. Diagram of the transformation process.

Using the theory of non-unimodular transformations described in [7], it can be deduced that T_a is a diagonal matrix, in which the diagonal elements are the loop step sizes. Therefore the bounds of the loops in OBIS are the same bounds as in NCBIS divided by the loop step size and the step size of all loops are 1. The loop iteration variables that appear in the bounds of the loops

must be multiplied by its corresponding step size. The matrix T_a and the bounds of the loops in the OBIS in our example are shown in Fig. 3(a) and 3(b) respectively.

$$\begin{array}{l}
T_a = \begin{bmatrix} B_i & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & B_j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{(a)}
\end{array}
\qquad
\begin{array}{l}
\text{do } i^B = \lceil (2-B_i)/B_i \rceil, \lfloor N/B_i \rfloor \\
\text{do } i = \max(1, i^B \cdot B_i), \min(i^B \cdot B_i + B_i - 1, N) \\
\text{do } j^B = \lceil (i - B_j + 1)/B_j \rceil, \lfloor N/B_j \rfloor \\
\text{do } j = \max(i, j^B \cdot B_j), \min(j^B \cdot B_j + B_j - 1, N) \\
\text{loop body} \\
\cdots \\
\text{enddo} \\
\text{(b)}
\end{array}$$

Fig. 3. (a) Matrix T_a . (b) Exact bounds of the loops in the OBIS.

The matrix $T = T_I \cdot T_a$ is used to compute the MCS of BTIS using the Fourier-Motzkin algorithm. The bounds of the MCS of BTIS in our example are shown in Fig. 4(a).

The exact bounds and step sizes of the loops are computed using the Hermite Normal Form (HNF) of T [16]. The HNF of T is a diagonal matrix because T_a is also a diagonal matrix and T_I is an interchange unimodular matrix. Since the HNF of T is diagonal, it can be directly computed in a trivial way. The diagonal elements of the HNF of T are the step sizes of the loops in the NCBIS interchanged according to T_I . The HNF of T in our example is shown in Fig. 4(b).

$$\begin{array}{l}
(2-B_i) \leq i^B \leq N \\
\max(2-B_j, i^B - B_j + 1) \leq j^B \leq N \\
\max(1, i^B) \leq i \leq \min(i^B + B_i - 1, j^B + B_j - 1, N) \\
\max(i, j^B) \leq j \leq \min(j^B + B_j - 1, N) \\
\text{(a)}
\end{array}
\qquad
\begin{array}{l}
\text{HNF}(T) = \begin{bmatrix} B_i & 0 & 0 & 0 \\ 0 & B_j & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{(b)}
\end{array}
\qquad
\begin{array}{l}
\text{do } i^B = \lceil (2-B_i)/B_i \rceil \cdot B_i, N, B_i \\
\text{do } j^B = \lceil \max(2-B_j, i^B - B_j + 1)/B_j \rceil \cdot B_j, N, B_j \\
\text{do } i = \max(1, i^B), \min(i^B + B_i - 1, j^B + B_j - 1, N) \\
\text{do } j = \max(i, j^B), \min(j^B + B_j - 1, N) \\
\text{loop body} \\
\cdots \\
\text{enddo} \\
\text{(c)}
\end{array}$$

Fig. 4. (a) Minimum Convex Space of BTIS. (b) The Hermite Normal Form of T . (c) Exact bounds of the loops in BTIS.

The exact bounds of the loops in BTIS are: a) the step sizes of the loops are the diagonal elements of the HNF of T , b) each upper bound is the same as in the MCS of BTIS and c) each lower bound is a ceiling function of the lower bound of the MCS of BTIS divided by its corresponding step size and all this expression multiplied by the same step size. The exact bounds and steps of the loops in BTIS are shown in Fig. 4(c).

We note that in the example used in this paper we deal with a diagonal HNF because we assume a null offset. With a non-null offset, matrix T_a and, therefore, the HNF of T are lower triangular matrices, but the bounds of the loops are also easy to compute. See [7] for further details about how the bounds of the MCS of BTIS are modified in the general case.

3.2 Register Level

To exploit data reuse at the register level after applying the loop interchange transformation, we need to fully unroll the most internal loops in the nest (loops that provide data reuse at this level) and to eliminate unnecessary loads/stores using scalar replacement. We will refer to the loops that we want to unroll as Unroll Candidates Loops (UCLs).

For example, the loop i of Fig. 5(a) is an UCL and to fully unroll this loop, it is necessary that i always executes the same number of iterations. After tiling, the number of iterations of the UCLs is not constant. However, the bounds of the UCLs always have: a) the iteration variable of the TI loop in one of the lower bound components and b) the iteration variable of the TI loop plus the tile size minus one in one of the upper bound components. Isolating this two loop bound components from the others, we will be able to fully unroll the UCLs. In the example we will isolate the bounds i^B and i^B+B_i-1 from the other bounds.

The loop i of Fig. 5(a) can be fully unrolled using conditional statements in the loop body. Nevertheless, the guards of the conditional statements are affine functions of the surrounding loops iteration variables, and the conditional statements cannot be moved outside the loop that surround the UCLs (loop k in Fig. 5(a)). So, it is not possible to apply scalar replacement to move the invariant references outside the innermost loop, and therefore there is no data reuse at the register level. To overcome this problem, we use Index Set Splitting (ISS) on the external loops in order to simplify the bounds of the UCLs, so that they can be fully unrolled.

In the example we split loop k into two new loops in such a way that in every iteration of one of the new loops the constraint $k \leq i^B$ holds, and in every iteration of the other loop the constraint $k > i^B$ holds. Graphically the iteration space defined by loops i and k is split as shown in Fig. 5(b). Now we can simplify the bounds of loop i in both new loop nests. In the loop nest where $k \leq i^B$ holds, the lower complex bound of i can be simplified to i^B ($\max(i^B, k) = i^B$). Similarly, in the loop nest where $k > i^B$ holds, the lower complex bound of i can be simplified to k ($\max(i^B, k) = k$). The resulting code after index set splitting is shown in Fig. 5(c). The UCL i of the first loop nest always executes B_i iterations and can be fully unrolled. The loop i of the second loop nest never executes a constant number of iteration and cannot be unrolled.

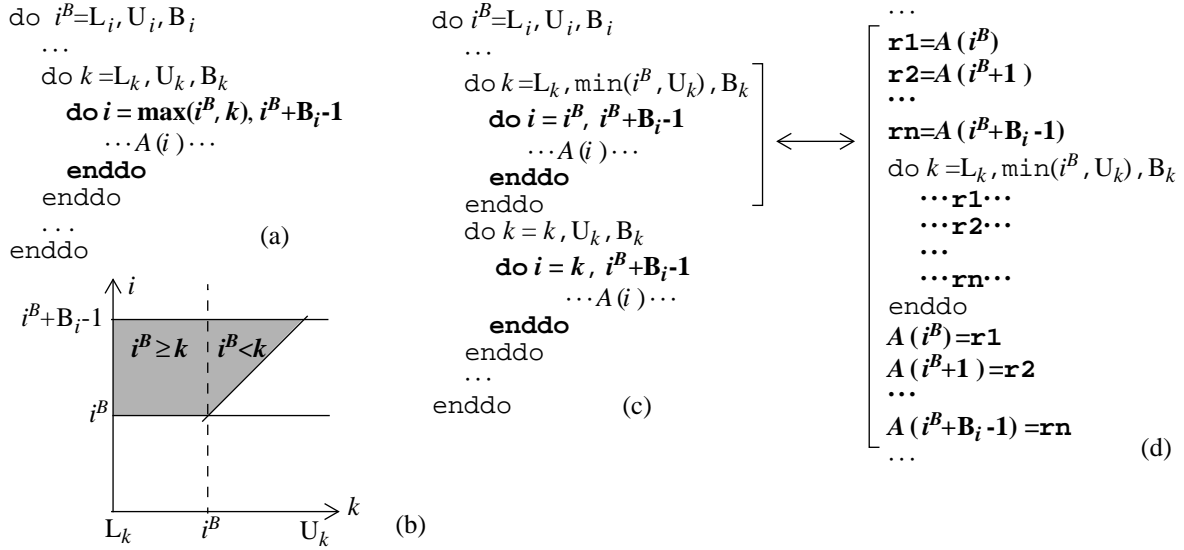


Fig. 5.(a) Loop nest, where i is an Unroll Candidate Loop.
 (b) Split the iteration space to be able to unroll loop i .
 (c) Loop nest after applying index set splitting.
 (d) Unrolled loop nest after applying unrolling and scalar replacement.

The order in which we apply ISS is very important to avoid processing a loop more than once and to avoid code explosion. We first apply ISS to the loops that we want to unroll (UCLs) from innermost to outermost (this ordering makes possible processing each loop only once). Then, ISS is applied to the rest of the loops from outermost to innermost (this second ordering avoids code explosion). See [8] for further details about the algorithm we use to apply ISS. Due to this order, it can happen that the UCLs are not directly surrounded by a loop. In this case it is necessary to apply loop distribution after ISS to be able to apply scalar replacement later on. In particular, we will distribute the loop that surrounds the UCLs. Figure 5(d) shows code of Fig. 5(a) after applying unroll and scalar replacement.

4 Experimental Results

To evaluate the effects of the proposed method, we have used 9 linear algebra algorithms. Their iteration spaces are 3-dimensional and not rectangular and in some of them the loops are not perfectly nested. Five of the algorithms are BLAS 3 operations (Basic Linear Algebra Subroutines). Table 1 contains a short description and the characteristics of each of the programs used in the evaluation. The column labeled "affine loop bounds" indicates the number of loops which loop bounds are affine functions of the surrounding loops iteration variables. The other loop bounds are integer or symbolic constants. In the column labeled "Ref" a reference of where the algorithms were extracted is given.

Ref	Name	Description	Characteristics	
			perfectly nested	affine loop bounds
[8]	MMtri	Triangular matrix product	Yes	2
[18]	LU	LU decomposition without pivoting	No	2
[6]	CHOL	Cholesky factorization	No	2
[19]	SOR	abstraction of 2-D hyperbolic PDE	Yes	2
BLAS3	SSYMM	symmetric matrix-matrix operation	No	1
	SSYRK	symmetric rank k update	Yes	1
	SSYR2K	symmetric rank 2k update	Yes	1
	STRMM	matrix-matrix operation	Yes	1
	STRSM	solve matrix equation	No	1

Table 1. Description and characteristics of each of the algorithms.

To perform the transformations, we have developed a tool where the technique proposed in this paper has been implemented. Our tool manipulates symbolic expressions. To all programs evaluated we apply tiling at the first level cache and at the register level. At both levels we apply tiling in two dimensions of the 3-dimensional iteration spaces.

The programs were compiled three times: one using only the native compiler (Native) with no previously restructuring transformation, a second one using the KAP preprocessor to restructure the code (KAP) and the third one using our tool, also as a preprocessor (Proposed). KAP is a commercial source to source preprocessor capable of restructuring code to exploit the different levels of the memory hierarchy. After the preprocessing step, we used the standard Fortran 77 compiler with the -O4 option to generate the final executables. The F77 compiler basically unrolls the innermost loop when the loop body performs a small number of operations to increase the instruction level parallelism.

We compare the performance (Mflops) obtained by each of these three compilations of the programs on a workstation with an ALPHA AXP 21064 at 200Mhz with a direct-mapped 8Kb first level cache, a direct-mapped 2Mb second level cache and 32 floating point registers.

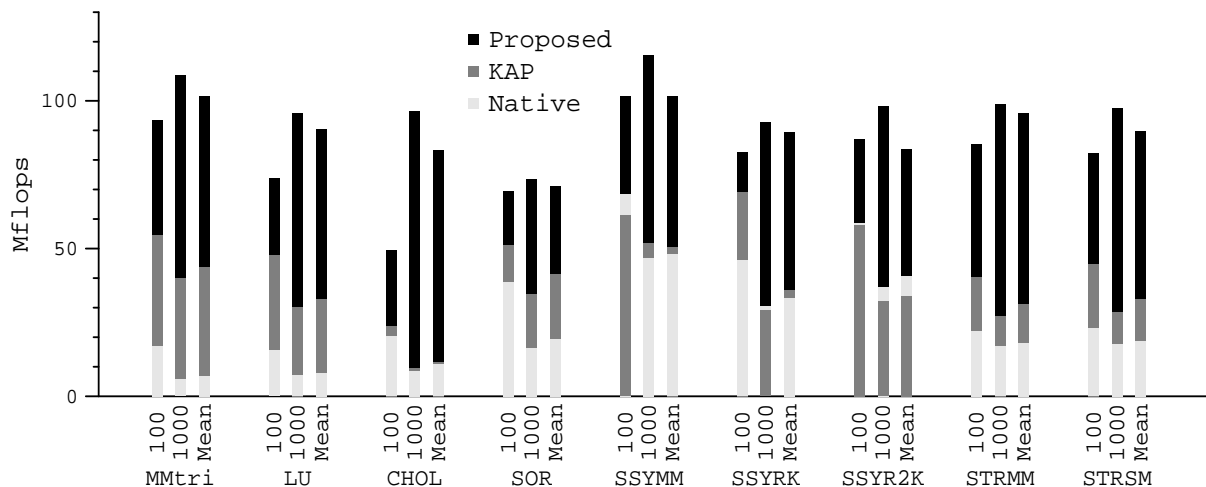


Fig. 6. Performance obtained by the three compilations (Native, KAP, Proposed) of each of the programs with matrix size $N=100$, $N=1000$, and the harmonic mean varying N from 100 to 1500.

4.1 Performance Results

Figure 6 shows the performance obtained by the three compilations of each program with matrix sizes equal to 100 and 1000 and also shows the harmonic mean of the performance obtained varying the matrix sizes (This mean is taken over 39 different matrix sizes varying from 100 to 1500). In Fig. 6 the three bars corresponding to the three compilations of each program are overlapped. In general the use of KAP as a preprocessor improves the performance over the native compiler, but in three cases (SSYR2K, SSYMM with $N=100$ and SSYRK with $N=1000$) it does not.

In SOR we need to apply loop skewing before tiling to convert the loops into a fully permutable loop nest [17]. The three compilations of SOR were done using the code once skewed.

Our tool used as a preprocessor always achieves the best performance and obtains speedups that vary from 1.73 to 7.14, depending on the program, over the KAP preprocessor. Table 2 shows the speedups obtained by our tool over the KAP and the native compiler in each program. The speedups were computed using the harmonic mean of Fig. 6.

Speedup	MMtri	LU	CHOL	SOR	SSYMM	SSYRK	SSYR2K	STRMM	STRSM
KAP	2.33	2.75	7.14	1.73	2.01	2.48	2.45	3.05	2.73
Native	14.88	11.59	7.73	3.65	2.11	2.68	2.06	5.30	4.80

Table 2. Speedups obtained by our tool over the KAP preprocessor and over the native compiler.

In Fig. 6 we note that the KAP preprocessor loses performance when the matrix size increases. This is due to the fact that the KAP preprocessor never exploits the cache level in this programs, since it is not able to produce an iteration space tiling when the bounds of the loops are affine functions (or compositions of affine functions) of the surrounding loops iteration variables (complex spaces). Nevertheless, in some cases the KAP preprocessor is able to interchange the loops to improve the data reuse.

At the register level, the KAP preprocessor basically unrolls the innermost loop (exploit register level only in one dimension) and applies scalar replacement. In only three of the programs the KAP was able to exploit the register level in two dimensions. Moreover, in a lot of programs it cannot apply scalar replacement effectively because it needs to interchange the loops and fails to do it because of the complexity of the bounds. The KAP uses scalar replacement only to move invariant references outside the innermost loop. So, in SOR it doesn't apply scalar replacement because there is no invariant references, but scalar replacement can be applied to reuse loads/stores in successive iterations of the inner loop.

The first transformation step performed in our tool consists in transforming a non-perfectly nested loop in a perfectly nested one before tiling the iteration space, using conditional statements. To see if the KAP preprocessor and the native compiler can take benefit from perfectly nested loops, we applied this transformation to all the programs with non-perfectly nested loops before compiling the programs. In all the cases, the KAP and the native compiler obtain less performance than without applying this transformation, because neither the native compiler nor the KAP preprocessor are able to eliminate conditional statements from the loop body. The KAP and Native results shown in Fig. 6 correspond to the compilation without applying this transformation.

5 Conclusions

In this paper we present a new unified method for code restructuring that aims at exploiting all levels of the memory hierarchy. Our proposal uses strip-mining and loop interchange to determine the tiles of all levels of the memory hierarchy. Afterwards we apply index set splitting to the register level to be able to fully unroll the inner loops after tiling and to eliminate unnecessary loads/stores, so we use efficiently the register level of the memory hierarchy.

We have also presented how to perform loop interchange in non-convex iteration spaces in a single step computing the bounds exactly and an order in which to perform index set splitting so that each loop in the nest will be processed only once and so that we avoid code explosion.

Finally, we have evaluated the performance obtained with the commercial KAP preprocessor and the performance obtained with our method in several linear algebra algorithms with complex iteration spaces. The performance obtained with our method is substantially higher than what the KAP preprocessor obtains. We are able to exploit the cache and register levels in complex iteration spaces.

Acknowledgments

This work was supported by the Ministry of Education and Science of Spain (CICYT TIC-0429/95).

References

- [1] A.J.C. Bik, H.A.G. Wijshoff. Implementation of Fourier-Motzkin Elimination. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.
- [2] D. Callahan, S. Carr, K. Kennedy. Improving Register Allocation for Subscripted Variables. International Conference on Programming Language Design and Implementation, June 1990, pp. 53-65
- [3] D. Callahan, J. Cocke, K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. Journal of Parallel and Distributed Computing, 1988, pp. 334-358.
- [4] S. Carr. Memory-Hierarchy Management. Ph.D. Dissertation, Rice University, Feb 1993.
- [5] S. Carr, K. Kennedy. Compiler Blockability of Numerical Algorithms. International Conference on Supercomputing, 1992, pp. 114-124
- [6] S. Carr, K. McKinley, C-W. Tseng. Compiler Optimizations for Improving Data Locality. International Conference on Architectural Support for Programming Languages and Operating Systems, Aug 1994, pp.252-262
- [7] A. Fernández, J.M. Llabería, M. Valero-García. Loop Transformation using non-unimodular matrices. IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 8, Aug 1995, pp. 832-840
- [8] M. Jiménez, J.M. Llabería, A. Fernández, E. Morancho. A Unified Transformation Technique for Multilevel Blocking. Technical Report UPC-DAC-1995-51, Dept. of Computer Architecture, Polytechnic University of Catalonia, Dec 1995.
- [9] K. Kennedy, K. M^cKinley. Optimizing for Parallelism and data Locality. International Conference on Supercomputing, July 1992, pp. 323-334
- [10] R. H. Khun. Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees. Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Feb 1980
- [11] P.M.W. Knijnenburg, A.J.C. Bik. On Reducing Overhead in Loops. Technical Report 94-40, Dept. of Computer Science, Leiden University, 1994.

- [12] M. Lam, E.Rothberg, M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63-74
- [13] J.J Navarro, A. Juan, M. Valero, J.M. Llabería, T. Lang. Multilevel Orthogonal Blocking for Dense Linear Algebra Computations. IEEE Computer Society TC on Computer Architecture Newsletter, Fall 1993, pp. 10-14
- [14] J.J Navarro, T. Juan, T. Lang. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations. International Conference on Supercomputing, July 1994, pp. 354-363
- [15] R. Schreiber, J. Dongarra. Automatic Blocking of Nested Loops, Technical Report, RIACS, NASA Ames Research Center and Oak Ridge Nat'l Laboratory, May 1990.
- [16] A. Schrijver. Theory of Linear and Integer Programming. Wiley-Interscience series in Discrete Mathematics, John Wiley and Sons, 1986
- [17] M. Wolf. Improving Locality and Parallelism in Nested Loops. Technical Report CSL-TR-92-538, Stanford University, Aug 1992.
- [18] M. Wolf, M. Lam. A Data Locality Optimizing Algorithm. International Conference on Programming Language Design and Implementation, June 1991, pp. 30-44
- [19] M. Wolf, M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Trans. on Parallel and Distributed System, Vol. 2, No. 4, October 1991, pp. 452-471
- [20] M.Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, 1996
- [21] M. Wolfe. More Iteration Space Tiling. International Conference on Supercomputing, 1989, pp. 655-664