A Unified Transformation Technique for Multilevel Blocking

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 only 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, unrolling and scalar replacement to this level. After applying strip-mining, the iteration space is non-convex. To perform in a single step the loop interchange in non-convex iteration spaces, we use non-unimodular matrices. The order proposed to perform index set splitting to the loops guarantees that each loop in the nest has to be processed only once and also avoids code explosion.

1 Introduction

To hide the details of the memory hierarchy from the user, several code transformation techniques have been developed. These techniques aim at exploiting the temporal and spatial locality properties of a program.

Iteration space tiling is a code restructuration technique used to reduce a program's data working set. Several proposals in the literature consider loop tiling on one, two or more levels of the hierarchy (registers, cache levels, TLB, virtual memory).Tiling an iteration space can be implemented using unroll & jam and scalar replacement at the register level[1][2], and applying strip-mining and loop interchange at all other levels of the hierarchy[3][6][9].

Tiling complex iteration spaces presents several problems. Applying strip-mining to a loop nest produces a non-convex iteration space since some of the loops in the nest will end up with a step different from 1. Therefore, the loop interchange needed after strip-mining can not be done using techniques based on unimodular transformation matrices [10]. Previous work on tiling, such as Wolf and Lam [10], uses ad-hoc methods to implement the loop interchange after strip-mining, and does not give a general algorithm. At the register level, S. Carr [2] uses pattern recognition techniques on the loop bounds to apply unroll and jam. Nevertheless, when the iteration space is complex, the loop bounds can not be matched by the patterns and no general algorithm to partition these complex iteration spaces into simpler ones, that could be recognized through patterns, is presented in [2].

Commercial compilers such as KAP from Kuck & Associates are not always able to produce an iteration space tiling when the loop bounds are affine functions of the surrounding loops iteration variables. These types of bounds are commonly found in linear algebra algorithms.

This paper presents a new unified method for simultaneously tiling the register and cache levels of the memory hierarchy. Because we do not use pattern recognition techniques, our method works for any iteration space having loop bounds defined as affine functions of the surrounding loops iteration variables. We will focus on the code transformation phase of tiling and we will assume that dependency analysis and locality analysis have already been performed [10][9]. Our algorithm uses strip-mining and loop interchange on all memory hierarchy levels (cache and registers) to determine the tiles as usual, and, afterwards, and due to the special characteristics of the register level, we also apply index set splitting, unrolling and scalar replacement to this level. The loop interchange in non-convex iteration spaces (spaces resulting after strip-mining) is done in a single step using non-unimodular matrices. We also use an order to perform index set splitting that guarantees that each loop in the nest has to be processed only once and also avoids code explosion.

2 Transformation Steps

We use the Multilevel Orthogonal Block (MOB) forms [7] to compute the tiles in each level of the memory hierarchy. The MOB forms provide maximum reuse of data in all levels simultaneously and their orthogonality property provides a simple method to optimize the form and the size of the tiles at each level. We will assume that the loops to be transformed are perfectly nested, fully permutable[8] and the loops bounds are affine functions of the surrounding loops iteration variables.
The steps we follow are:

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

1. Apply strip-mining to all loops selected using the MOB forms.
2. Interchange loops as determined by the MOB forms.

At the register level:

3. Apply index set splitting repeatedly until being able to unroll those loops that provide data reuse at the register level.
4. Distribute the surrounding loop of the loops we want to fully unroll.
5. Fully unroll all innermost loops.
6. Apply scalar replacement to all innermost loops [2].

Due to the lack of space, in this paper we will focus only at the register level. In [5] we explain how to perform strip-mining and interchange in non-convex iteration spaces in a single step.

2.1 Register Level

To exploit data locality 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 apply scalar replacement. We will refer to the loops that we want to unroll as Unroll Candidates Loops (UCLs).

Scalar replacement [2] finds opportunities for reuse of subscripted variables and replaces the references involved by references to temporal scalar variables; we use scalar replacement for determining array elements that are loop invariant with respect to the iteration variable of the innermost loop.

The loop i of Fig.1(a) is an UCL and can be fully unrolled using conditional statements in the loop body as shown in Fig.1(b). Loop i inside the if-part can be fully unrolled since it always executes the same number of iterations. The choice between the unrolled and non-unrolled version is performed at runtime by the if-test. The overhead of the if-test seems to be equivalent to the overhead of the `max` function in Fig.1(a). Nevertheless, the problem of using conditional statements is that, in the general case, the conditional statement cannot be moved outside the loop that surround the UCLs (loop k in Fig.1(b)) and it is not possible to apply scalar replacement to this loop.

In general, the bounds of the UCLs are affine functions of the surrounding loops iteration variables, and it is not possible to apply scalar replacement to the innermost loop, and therefore there is no data reuse at the register level.To overcome

```
do 10 $i^B$=$L_i$, $U_i$, $B_i$
 ...
   do 10 $k$=$L_k$, $U_k$, $B_k$
      do 10 $i$ = max($i^B$, $k$), $i^B$+$B_i$-1
        ...$A(i)$...
10 continue
```

(a)

```
do 10 $i^B$=$L_i$, $U_i$, $B_i$
 ...
   do 10 $k$=$L_k$, $U_k$, $B_k$
   if ($i^B$.ge.$k$) then
       do 20 $i$= $i^B$, $i^B$+$B_i$-1
         ...$A(i)$...
20   continue
   else
       do 30 $i$ = $k$, $i^B$+$B_i$-1
         ...$A(i)$...
30   enddo
   endif
10 continue
```

(b)

```
do 10 $i^B$=$L_i$, $U_i$, $B_i$
 ...
   do 20 $k$=$L_k$, min($i^B$, $U_k$), $B_k$
      do 20 $i$ = $i^B$, $i^B$+$B_i$-1
        ...$A(i)$...
20 continue
   do 10 $k$ = $k$, $U_k$, $B_k$
      do 10 $i$ = $k$, $i^B$+$B_i$-1
        ...$A(i)$...
10 continue
```

(c)

Fig. 1. (a) Example of loop nest. (b) Code using conditional statements. (c) Loop nest after applying ISS.

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.

For example, to unroll loop i in Fig.1(a), it is necessary that i always executes the same number of iterations. Isolating the bounds i^B and i^B+B_i-1 from the other bounds, we will achieve this goal.

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. 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.1(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.

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 [5] for further details on how we 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 2(a) shows code of Fig.1(a) after applying unroll and scalar replacement.

3 Results and Conclusions

We used a triangular matrix product (MxM) algorithm and a rank 2k update (SYR2K) algorithm to evaluate the effects of the new method proposed. The matrices in the SYR2K algorithm are banded and stored in compact form, therefore the complexity of the loop bounds in SYR2K is higher than in the triangular matrix product algorithm.

The two algorithms are compiled first without applying any restructuring transformation, then using the KAP compiler to restructure the code, and finally applying our new method. We compare the performance (Mflops) obtained by each of these three versions of the algorithms on an ALPHA AXP 21064 (direct-mapped 8Kb cache memory and 32 floating point registers). In the results presented in this paper, we considered registers and first level cache.

In the triangular MxM algorithm the KAP compiler generates different code depending on the initial loop ordering so we have selected the ordering that yields the

```
      do 10 $i^B$=$L_i$, $U_i$, $B_i$
        ...
        r1=$A(i^B)$
        r2=$A(i^B+1)$
        ...
        rn=$A(i^B+B_i-1)$
        do 20 $k$=$L_k$,min($i^B$, $U_k$), $B_k$
          ...r1...
          ...r2...
          ...
          ...rn...
20      continue
        $A(i^B)$=r1
        $A(i^B+1)$=r2
        ...
        $A(i^B+B_i-1)$ =rn
        do 10 $k$ = $k$, $U_k$, $B_k$
          do 10 $i$ = $k$, $i^B+B_i-1$
            ...$A(i)$...
10    continue                    (a)
```

(b)

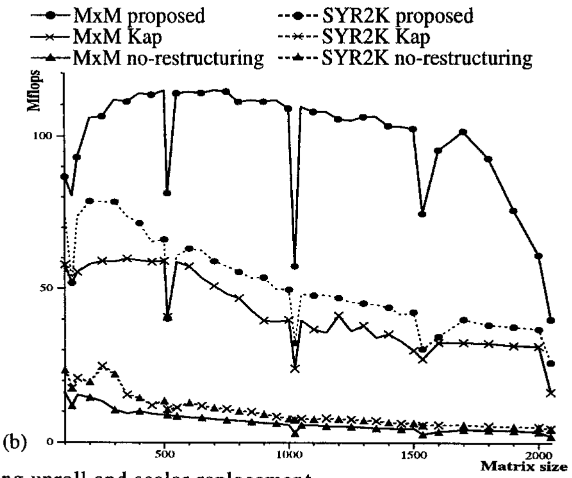

Fig. 2. (a) Loop nest after applying unroll and scalar replacement.
(b) Performance obtained by each method for different matrix sizes.

best performance. Figure 2(b) (solid lines) shows how our proposal doubles the Mflops obtained by the KAP compiler. For SYR2K the KAP compiler is only able to apply scalar replacement without unrolling. Figure 2(b) (dotted lines) shows the Mflops obtained with a fixed band of 50 and different matrix sizes for the three versions of the SYR2K algorithm. The proposed method obtains a speedup of around 4 over the KAP compiler.

In this paper we have presented a new unified method for code restructuring that aims at exploiting all levels of the memory hierarchy. The performance obtained with the method presented in this paper is substantially higher than the performance obtained by commercial compilers. Moreover, the relative improvement obtained with our method increases as the complexity of the loop bounds (such as in SYR2K) also increases.

Acknowledgments

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

References

1. D. Callahan, S. Carr, K. Kennedy. Improving Register Allocation for Subscripted Variables. Int. Conf. on Programming Language Design and Implementation, June 1990, pp. 53-65
2. S. Carr. Memory-Hierarchy Management. Ph.D. Dissertation, Rice University, Feb 1993.
3. S. Carr, K. McKinley, C-W. Tseng. Compiler Optimizations for Improving Data Locality. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Aug 1994, pp.252-262
4. 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
5. M. Jiménez, J.M. Llabería, A. Fernández, E. Morancho. A Unified Transformation Technique for Multilevel Blocking. TR. UPC-DAC-1995-51, Dept. of Computer Architecture, Polytechnic University of Catalonia, Dec 1995.
6. M. Lam, E.Rothberg, M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63-74
7. J.J Navarro, T. Juan, T. Lang. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations. Int. Conf. on Supercomputing, July 1994, pp. 354-363
8. M. Wolf. Improving Locality and Parallelism in Nested Loops. Technical Report CSL-TR-92-538, Stanford University, Aug 1992.
9. M. Wolf, M. Lam. A Data Optimizing Algorithm. Int. Conf. on Programming Language Design and Implementation, June 1991, pp. 30-44
10. 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
11. M. Wolfe. More Iteration Space Tiling. Int. Conf. on Supercomputing, 1989, pp. 655-664