A General Algorithm for Tiling the Register Level

M. Jiménez, J. M. Llabería, A. Fernández and E. Morancho
Departamento de Arquitectura de Computadores
Universitat Politècnica de Catalunya
{marta,llaberia,agustin,enricm}@ac.upc.es

1. ABSTRACT

Tiling is a well-known loop transformation that can be used to exploit data reuse at the register level and to improve a program's ILP. Previous work on tiling and also commercial compilers are able to perform tiling for the register level in more than one dimension when the iteration space is rectangular. However, they either cannot handle or can only handle limited cases of non-rectangular iteration spaces. Non-rectangular iteration spaces[1] are commonly found in linear algebra algorithms or can arise as a result of applying previous transformations such as loop skewing. In this paper we present a new general algorithm to perform tiling for the register level in more than one dimension in both rectangular and non-rectangular iteration spaces. Our method uses index set splitting to distinguish loop nests that traverse *boundary* tiles of the tiled iteration space from loop nests that traverse *non-boundary* tiles. We evaluate our method using as benchmarks typical linear algebra algorithms having non-rectangular iteration spaces. Results measured on both ALPHA 21064 and MIPS R10000 machines show that our method achieves speedups in the range of 1.11 to 5.96 over commercial compilers and preprocessors able to perform optimizing code transformations.

2. INTRODUCTION

Modern processors issue multiple instructions per cycle to exploit instruction level parallelism (ILP) and use several memory levels to reduce the average memory access time. To achieve high performance in these processors, the compilers have to be able to exhibit a program's ILP and to increase data locality to take full advantage of the architecture and to minimize the data movement between different levels of the memory hierarchy. Being able to exploit data reuse at the register level is extremely important in today's superscalar microprocessors, since the register level directly feeds the processor functional units and, if the register level is not properly exploited, then the number of first level cache ports bounds processor performance. Much research has been devoted to exploit data reuse at the cache level [5][12][20] and less attention has been paid to the register level [1][2].

[1] We refer as a "non-rectangular" iteration space to an iteration space defined by loops whose bounds are compositions (`max` or `min`) of affine functions of the surrounding loops iteration variables. This includes iteration spaces whose bounds are not parallel to the iteration space axes.

Several code transformation techniques have been developed to exploit a program's ILP [3][13] and/or to improve the memory hierarchy utilization [21][22]. Compiler transformations such as inner unrolling [23] and software pipelining [13][17] are used to improve ILP, but do not properly exploit data reuse at the register level. Another compiler transformation is *unroll & jam* [1][2], which is able to both improve ILP and exploit data reuse at the register level in the unrolled dimension.

It is well known that exploiting data reuse in more than one dimension of the iteration space, whenever possible, improves the performance of the memory hierarchy [14][15]. *Tiling* (also called *blocking*) [18][24] is a well-known loop transformation that can be used to exploit data reuse at the register level in several dimensions. Futhermore, tiling for the register level has the desirable property that it always increases ILP [11]. Thus, tiling has the potential of outperforming the other code transformations mentioned above [11].

Previous work on tiling [2][19], and also commercial compilers and preprocessors, either cannot handle or can only handle limited cases of non-rectangular iteration spaces. Tiling arbitrary non-rectangular iteration spaces is not generally considered because it is not trivial [19]. We note that unroll & jam can be applied in multiple dimensions and, in rectangular iteration spaces, it is the same transformation as register tiling. However, unroll & jam can only be applied to limited cases of non-rectangular spaces[2].

In this paper we present a new general method to perform tiling for the register level that blocks in several dimensions of the iteration space, improving the performance of the memory hierarchy and the ILP to take full advantage of the target architecture. This work extends upon previous work by handling arbitrary non-rectangular iteration spaces.

To evaluate the performance obtained by our method, we use as benchmarks typical linear algebra algorithms having non-rectangular iteration spaces. We compare our proposal against commercial compilers and preprocessors able to perform optimizing code transformations on two different superscalar microprocessors (ALPHA 21064 and MIPS R10000).

This paper is organized as follows. Section 3 presents our general method to perform tiling at the register level. In section 4 we present the evaluation of our method through some experimental results. In section 5 we present the previous work related to memory hierarchy management and finally, in section 6, we draw some conclusions.

3. TILING FOR THE REGISTER LEVEL

In this section we will describe the transformation steps carried out by our method to perform tiling for the register level. The method consists in a combination of well-known transformations: strip-mining, interchange, index set splitting, unrolling and scalar replacement.

We assume that the loop bounds are `max`/`min` functions of affine functions of the surrounding loops iteration variables. We also

assume that the original loop nest is fully permutable and perfectly nested. However, we note that (a) non fully permutable loop nests could be turned into fully permutable nests by applying a loop skewing transformation [19][21] and (b) non perfectly nested loops can be converted into perfectly nested loops using a code sinking transformation [23]. To avoid the execution time overhead introduced by the code sinking transformation, it must be undone after tiling. Finally, we also assume that a previous analysis to decide (a) which loops are the best ones to be tiled, (b) the tile size in each dimension and (c) the order of the loops that delivers the best performance, has already been performed (see [9][11]).

Our method applies first loop tiling to divide the iteration space defined by the loop structures into regular tiles. Loop tiling can be implemented using strip-mining and loop interchange [7][10][21]. Strip-mining is used to partition the iteration space and the loop interchange is used to order the loops in such a way that (a) the loops that step between tiles become the outer loops (*tile* loops) and (b) the loops that step points within a tile become the inner loops (*element* loops).

Registers are only addressable using absolute addresses (register number). Therefore, after tiling the iteration space, it is necessary to fully unroll the loops that step the points inside the register tiles (the *element* loops). A loop is fully unrolled by replicating the loop body as many times as the loop bounds indicate, changing the iteration variable that appears in the unrolled loop body by its different values and eliminating the do-loop statement; a new loop body is obtained.

After tiling a loop nest, we obtain a new single loop nest that traverses all tiles that cover the entire original iteration space. We refer as a *boundary* tile to a tile whose intersection with the original iteration space is not equal to the tile. In *boundary* tiles only some points inside the tiles are traversed (in *non-boundary* tiles all points are traversed). In rectangular iteration spaces, the intersection of a *boundary* tile with the original iteration space is always a rectangular space. However, in non-rectangular iteration spaces, this intersection can be a non-rectangular space. This fact prevents directly fully unrolling the *element* loops after tiling a non-rectangular iteration spaces.

We will distinguish in the tiled code loop nests that traverse *boundary* tiles from loop nests that traverse *non-boundary* tiles. Thus, after tiling the original loop nest, our method applies index set splitting [23] repeatedly with the goal of isolating a region (partition) of the tiled iteration space that contains only *non-boundary* tiles. The *element* loops of the loop nest that traverses this partition can be fully unrolled since they execute a constant number of iterations. However, not all the *element* loops of loop nests that traverse partitions containing *boundary* tiles can be fully unrolled.

At last, after applying index set splitting, scalar replacement [1] [2] is used to eliminate redundant loads and stores in the loop body. Scalar replacement finds opportunities for reuse of subscripted variables and replaces the references involved by array references to temporal scalar variables.

We note that our method can be used in the context of multilevel tiling. If exploiting data reuse in several levels of the memory hierarchy is desired, then multilevel tiling is applied in the first step [10], and all other steps remain unchanged.

Next, we first illustrate with an example all the transformation steps performed by our method, paying special attention to the Index Set Splitting step. Thereafter, we formalize the complete algorithm to apply index set splitting.

3.1 How to Apply Index Set Splitting

We will use the triangular matrix product algorithm to clarify all steps performed by our method. We note that this algorithm, shown in Fig. 1a, has a non-rectangular iteration space. The first step of our method consists in tiling the original loop nest [10][21]. Suppose that the loops to be tiled for the register level are loops *j* and *i* and the tile sizes are R1 and R2, respectively. For simplicity, we suppose that N is multiple of R1 and R2. The code after tiling is shown in Fig. 1b.

We will refer to the loops that we want to unroll (the *element* loops) as Unroll Candidate Loops (UCLs). In Fig. 1b, the UCLs are loops *j* and *i*. In the next step of our method, index set splitting is used to isolate a partition of the tiled iteration space where the UCLs iterate exactly as many times as the tile size in their dimension. In this partition, the UCLs can be fully unrolled.

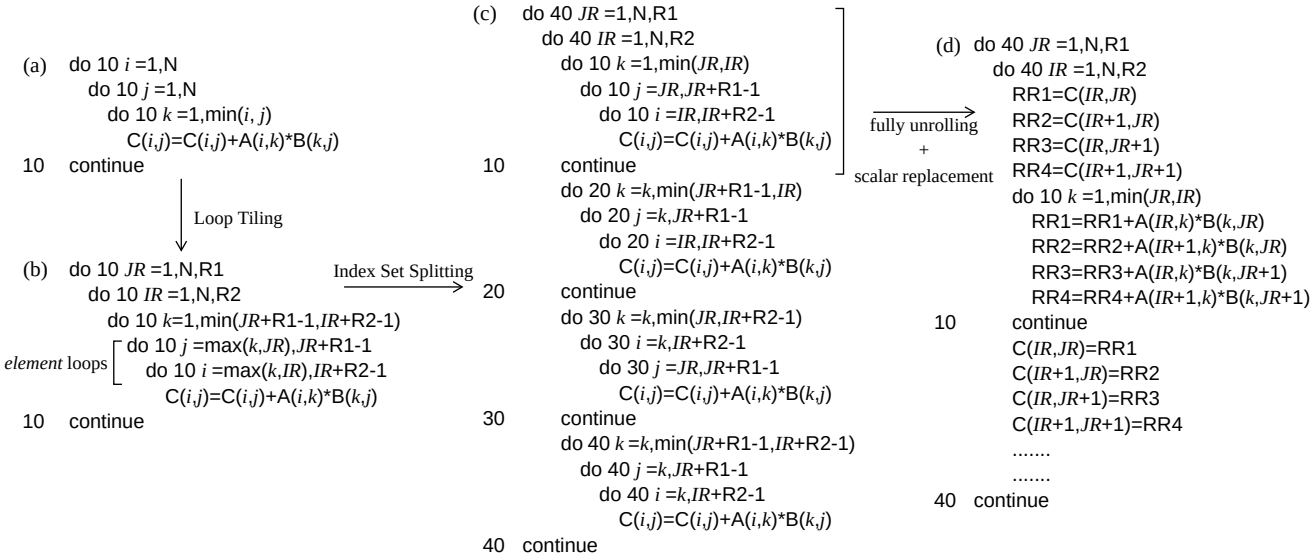

Figure 1. (a) Original code (form *ijk*) of triangular matrix product. (b) Code after tiling for the register level. (c) Code after applying index set splitting repeatedly. (d) Piece of the code after fully unrolling and applying scalar replacement (R1=R2=2).

3.1.1 Index Set Splitting

Let *IR* be the iteration variable of a *tile* loop and let R2 be the tile size in this dimension. Its associated *element* loop (that is, an UCL) always has *IR* and *IR*+R2-1 as one lower and upper bound component, respectively. We want to achieve a partition of the tiled iteration space, where those bound components are the *effective*[1] bounds of the UCL.

Index Set Splitting (ISS) splits a loop into two new loops, where each new loop iterates over non-intersecting partitions of the original loop. We use Index Set Splitting to split outer loops (with respect to the UCL) with the goal of dividing the tiled iteration space into partitions.

The loop bounds of all UCLs determine the set of all conditions that must hold in the partition where all UCLs can be fully unrolled. In our example, the UCLs *j* and *i* (whose lower bounds are max(*k*,*JR*) and max(*k*,*IR*), respectively) could be fully unrolled if the conditions $JR \geq k$ and $IR \geq k$ hold. Thus, we will apply ISS with each of these conditions.

We start dealing with the condition $IR \geq k$. First, we solve the condition $IR \geq k$ for the innermost loop whose iteration variable appears in the inequality (loop *k*); this loop will be the loop to be split. The new inequality ($k \leq IR$) will be referred to as *restriction*[2]. Second, we split loop *k* using restriction $k \leq IR$ into two new loops, in such a way that in one of the new loop the restriction $k \leq IR$ always holds and in the other does not.

Figure 2a shows the loop nest before applying ISS, with the bound that has generated the restriction and the loop to be split marked in bold. Figure 2b shows the code after applying ISS to loop *k* and in Fig. 2c we can see graphically how the iteration space defined by loops *i* and *k* is split.

After applying ISS to loop *k*, the bounds of loop *i* in both partitions can be simplified. In the partition where $k \leq IR$ holds (partition 1), the lower bound of *i* can be simplified to *IR* (max (*k*,*IR*) = *IR*). In a similar way, in the partition where $k \leq IR$ never holds (partition 2), the lower bound of *i* can be simplified to *k* (max (*k*,*IR*) = *k*). In Fig. 2b the main changes on the loop bounds are marked in bold.

Now loop *i* of partition 1 always executes R2 iterations and can be fully unrolled. Loop *i* of the second partition never executes a constant number of iterations and cannot be fully unrolled. Loop *i* of partition 2 is no longer an UCL.

[1] An *effective* bound is the bound component that results after evaluating a composition (`max` or `min`) of bound components.

[2] A *condition* and a *restriction* refer to different but equivalent expressions. A *restriction* is a *condition* after solving it for the innermost loop iteration variable.

We continue applying index set splitting repeatedly to the partition where all conditions that have been previously applied hold, until we achieve a partition where all UCLs can be fully unrolled. In Fig. 2b we have to apply ISS again to partition 1 with the condition $JR \geq k$ to be able to also fully unroll UCL *j*.

After isolating the partition where all UCLs can be fully unrolled, we obtain other partitions that only contain *boundary* tiles. In certain *boundary* tiles, some (but not all) *element* loops can also iterate exactly as many times as the tile size in their dimension and, therefore, they can also be fully unrolled. In partition 2 of Fig. 2b, for example, loop *j* can be fully unrolled if ISS is applied again to this partition. Our method deals also with all these partitions. In these partitions, it is sometimes necessary to perform an interchange to make sure the UCLs become innermost loops before fully unrolling them.

Figure 1c shows the final code after applying ISS repeatedly to all partitions. The first loop nest traverses the partition containing only *non-boundary* tiles and, therefore, loops *j* and *i* can be fully unrolled (they always execute R1 and R2 iterations respectively). The other three loop nests traverse partitions containing *boundary* tiles. However, in the second and third loop nests some, but not all, *element* loops can be fully unrolled (loops *i* and *j*, respectively).

3.1.2 Eliminate Redundant Loads and Stores

In our final step, we fully unroll the loops and apply scalar replacement to eliminate redundant loads and stores in each partition. The final resulting code after all steps is shown in Fig. 1d (assuming R1=R2=2). For the sake of brevity, we only show the first loop nest and we use scalar replacement to only move invariant references outside the innermost loop.

3.2 Index Set Splitting Algorithm

In this section we formalize how to apply index set splitting repeatedly and we give analytical expressions both for the complexity of our method and for the amount of code generated. All discussion in this section assumes that loop tiling has already been performed.

Initially, we want to isolate a partition of the tiled iteration space where the *effective* bounds of all UCLs are: the iteration variable of the *tile* loop in the lower bound and the iteration variable of the *tile* loop plus the tile size minus one in the upper bound. In this partition of the iteration space a set *C* of conditions hold. The conditions are those ensuring that the lower (upper) bound component we want to be the *effective* bound in each UCL is greater (smaller) than the other lower (upper) bound components of the loop bound. Thus, from the bounds of each UCL in the tiled code we obtain a set of conditions C_v and the bounds of all UCLs determine the set *C* of all conditions that must hold in the partition where all UCLs can be fully unrolled.

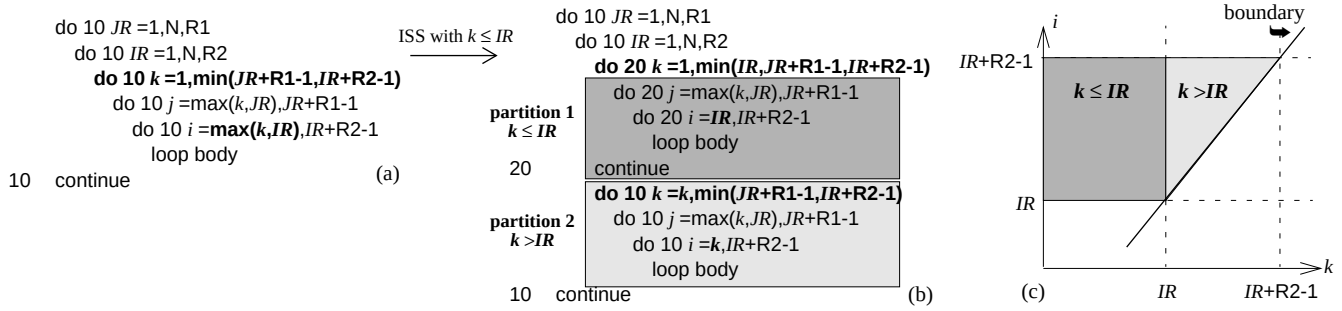

Figure 2. Loop nest a) before applying ISS, b) after applying ISS with restriction $k \leq IR$. c) Iteration space defined by loops *i* and *k*.

Let n be the total number of loops in the loop nest after tiling, numbered from 1 to n from the outermost to the innermost position, and let d be the number of UCLs. The d UCLs of the n total loops are always in the innermost positions after tiling; then, the set *C* of all conditions is:

$$C = \bigcup_{v = n-d+1}^{n} \{C_v\}$$

Let r_v+1 and s_v+1 be the number of lower and upper bound components of UCL i_v, respectively. One of the r_v+1 and one of the s_v+1 bound components are the *effective* bounds. The total number of conditions we have to deal with initially is:

$$|C| = \sum_{v = n-d+1}^{n} (r_v + s_v)$$

Given one condition, the loop to be split is the innermost loop whose iteration variable appears in the condition and it is always a loop that surrounds the UCL whose bounds have generated this condition. Let i_p be the loop to be split. We obtain the restriction used to perform ISS by solving the condition for i_p. Two types of restrictions can be obtained depending on the sign of the coefficient of i_p: when the coefficient is negative we have a *lower restriction* and when it is positive we have an *upper restriction*.

The loop to be split is divided into two consecutive loops that iterate over non-intersecting partitions of the iteration space. The type of the restriction determines in which of the two partitions the restriction holds. In both partitions there is a lower or upper bound component of the loop that has generated the condition that is redundant and can be removed.

In the partition where the restriction holds, the loop that has generated the condition is still an UCL, because it can be fully unrolled if we repeatedly apply ISS to this partition until all the necessary conditions hold. Nevertheless, in the other partition this loop is no longer an UCL, because it does not have any longer one of the bound components that we want to be an *effective* bound.

We note that the loop to be split can be an UCL. When we apply ISS a new bound component appears in the loop being split. If that loop is an UCL we will have to add a new condition to the set *C* of conditions. Moreover, in the partition where the restriction does not hold, both the loop that has been split and the loop that has generated the condition are no longer UCLs.

3.2.1 Processing Order

The order in which we deal with each restriction, that is, the order in which we split the loops, is very important to avoid processing a loop more than once and to reduce code expansion.

If we split the loops from outermost to innermost, we reduce code expansion[1]. However, when we apply ISS to an UCL we will have to deal with a new restriction that will split an outer loop. This would induce the need to split loops that have already been processed, leading to repeated processing of some loops. On the other hand, if we split the loops from innermost to outermost, we process each loop only once since the new bound components appear on outer loops which are still pending to be processed.

Taking into account that the restrictions that split UCLs are the only ones that can introduce new restrictions we propose the following order to deal with restrictions: We deal first with restrictions that split UCLs from innermost to outermost. In this way we process each loop only once. Second, we deal with restrictions that split loops that are not UCLs from outermost to innermost. This reduces code expansion (see [9] for further details).

[1] Every time ISS is applied, the loop body of the loop being split is replicated.

```
Algorithm
 INPUT: OL /* loop nest after tiling */
        n  /* total number of loops in the loop nest OL */
        d  /* total number of UCLs (the innermost loops) */
 OUTPUT: /* transformed loop nest */

 LB={OL} /* list of loop nests waiting for to be dealt with */
 while (LB is not empty)
 { AN=first loop nest of LB; /* AN is the active loop nest */
   LB=LB-{AN};

   Create the sorted list LL of restrictions determined
   by the still UCLs in AN;
   while (LL is not empty)
   { R=first restriction of list LL;
     p=loop to be split according to restriction R;
     Split loop ip of AN according to R;
     if (R has the form ip ≤ u')/* upper restriction */
     { AN=1st partition;
       LB=LB+{2nd partition};
     }
     else /* R has the form ip ≥ l'; lower restriction */
     { AN=2nd partition;
       LB=LB+{1st partition};
     }
     if (p > n-d)   /* the loop ip is an UCL */
       Insert new restriction into sorted list LL;
     LL=LL-{R};
   }

   /* UCLs must be in the innermost positions */
   Interchange loops of AN if necessary;
   Unroll the UCLs of AN;
 }
endAlgorithm
```

Figure 3. Code of the complete ISS algorithm.

3.2.2 The Index Set Splitting Algorithm

The complete algorithm for ISS is shown in Fig. 3. We call AN ("Active Nest") the loop nests where we want to fully unroll the UCLs, that is, the loop nest where the restrictions are applied. Initially the AN is the loop nest after tiling. We create the list of restrictions sorted according to the order described above and we deal one by one with all of them. Each time we apply ISS according to a restriction, we save the partition where the applied restriction does not hold in the list of loop nests pending to be processed. When a restriction splits a UCL a new restriction appears. In this case, we insert the new restriction in the sorted list of restrictions we are dealing with.

In the partitions where some restrictions do not hold, not all *element* loops can be fully unrolled because some of them are no longer UCLs. However, we can apply ISS again to these partitions to try to unroll the *element* loops that are still UCLs. When dealing with these partitions it is sometimes necessary to perform a loop interchange transformation to make sure the UCLs become innermost loops before fully unroll them. This interchange transformation can be directly performed because no inner *element* loop can have bound components that are affine functions of the still UCLs.

At the end of the process there will be one partition where all the *element* loops can be fully unrolled, some partitions where some, but not all, *element* loops can be fully unrolled and some partitions where no loops can be fully unrolled.

3.2.3 Complexity and Code Expansion

We measure the complexity in number of times ISS has to be performed and the amount of code generated in number of loop nests generated. For simplicity, the expressions we give in this section are developed for two UCLs in the loop nest, but they can be easily extended for any number of UCLs.

The number of times that our algorithm performs ISS depends on the number of bound components of the UCLs in the tiled code. Let R be the number of (upper and lower) bound components of the outermost UCL, and let S+M be the number of bound components of the innermost UCL, where the M bound components are affine functions of the outer UCL and the S bound components are not. (Neither R nor S+M contain the bound components that we want to be the *effective* ones).

The total number of times that our algorithm performs ISS (N_{iss}) to achieve that the UCLs in all partitions can be fully unrolled is between the range: $M+R+S+R*S \le N_{iss} \le M+R+M+S+(R+M)*S$, and the total number of loop nests generated is $N_{total} = (M+1)*(N_{iss}-M+1)$. The development of these expressions is explained in [9].

4. EVALUATION

In this section we will present the performance results obtained by tiling for the register level, and we will compare it on two different superscalar microprocessors against the native compilers and against the commercial KAP[1] preprocessor. Since this work extends upon previous work on register tiling by handling arbitrary non-rectangular iteration spaces, we use as benchmarks typical linear algebra algorithms having non-rectangular iteration spaces. We will first describe our evaluation process and then present the performance results.

4.1 Evaluation Process

4.1.1 Benchmark Programs

As benchmark programs, we have used 9 linear algebra algorithms having non-rectangular, 3-dimensional iterations spaces. Table 1 contains a short description and the characteristics of each of them. Column labeled “Ref” indicates from where the algorithms were extracted. The fourth column indicates whether the loops being transformed were perfectly nested or not. As pointed out in section 3, for those programs having non-perfectly nested loops, we transformed them into a perfectly nested version using a code sinking transformation that is undone after loop tiling. Column labeled “affine bounds” indicates the total number of bound components in the original code that are affine functions of the surrounding loops iteration variables. The other bound components are integer or symbolic constants. Column labeled “section analysis” indicates wether it is necessary to perform an array section analysis [4][6] to determine if the loop is fully permutable or not. In one case (program SOR), we were forced to apply loop skewing to convert the loops into a fully permutable loop nest [19]. All results presented for SOR were measured using the code once skewed.

[1] KAP is a commercial source to source preprocessor from Kuck and Associates capable of restructuring code to exploit both the different levels of the memory hierarchy and the program’s ILP.

Ref	Name	Description	perfect nested	affine bounds	section analysis
[8]	MMtri	Triangular matrix product	Yes	2	No
[20]	LU	LU decomposition without pivoting	No	2	Yes
[5]	CHOL	Cholesky factorization	No	2	Yes
[21]	SOR	abstraction of 2-D hyperbolic PDE	Yes	4	No
BLAS3	SSYMM	symmetric matrix-matrix operation	No	1	No
	SSYRK	symmetric rank k update	Yes	1	No
	SSYR2K	symmetric rank 2k update	Yes	1	No
	STRMM	matrix-matrix operation	Yes	1	Yes
	STRSM	solve matrix equation	No	1	Yes

Table 1: Description and characteristics of several linear algebra algorithms.

4.1.2 Code Generation

To perform tiling for the register level, we have developed a tool that implements our technique. To all programs evaluated we always tile two dimensions of the 3-dimensional iteration spaces [15]. The algorithm used to determine the best tiled loops for the register level is beyond the scope of this paper (see [9]). However, we note that in our benchmarks we select the tiled loops that provide more temporal data locality for register reuse. We do not consider explicitly the amount of ILP that can be achieved, since tiling at more than one dimension for the register level, regardless of the loops being tiled, always achieves a reasonable amount of ILP [11]. The tile sizes were chosen taking into account the available number of machine registers in order to reduce the register pressure and not overly constrain the job of the register allocator of the native compilers.

4.1.3 Target Architectures

We used the MFLOP/s metric as our indicator of performance. All our measurements were taken on a uniprocessor system with an ALPHA 21064 processor and on a single R10000 processor of a multiprocessor system (Power Challenge). The two different architectures are shortly described in Table 2.

Architecture	MHz	Regs		L/S cycle	rd/wr regs ports	L1	L2	TLB entries	issue
		int	fp						
ALPHA 21064	200	32	32	1/1	3/1	8Kb	2Mb	32	in-order
MIPS R10000	~200	32	32	1/1	5/3	32Kb	1Mb	64	out-of-order

Table 2: Memory hierarchy characteristics of the architectures ALPHA AXP 21064 and MIPS R10000.

4.2 Performance Results

To compare tiling for the register level against other optimizing code transformations performed by commercial compilers and preprocessors, we evaluate three different versions of each program: one is the original version (ORI) with no previously restructuring transformation, a second one generated using the KAP preprocessor to restructure the code (KAP) and the third one generated using our tool, also as a preprocessor, to tile for the register level (TRL).

After generating the different versions for each program, we used the standard Fortran 77 compiler to generate the final executables. The F77 compiler was used with the scalar optimizations recommended by the manufacturer turned on (-O4 on the ALPHA and -O3 on the MIPS). It is worth noting that, at these optimization levels, the F77 compiler unrolls the innermost loop when the loop body has a small number of operations in order to increase the instruction level parallelism.

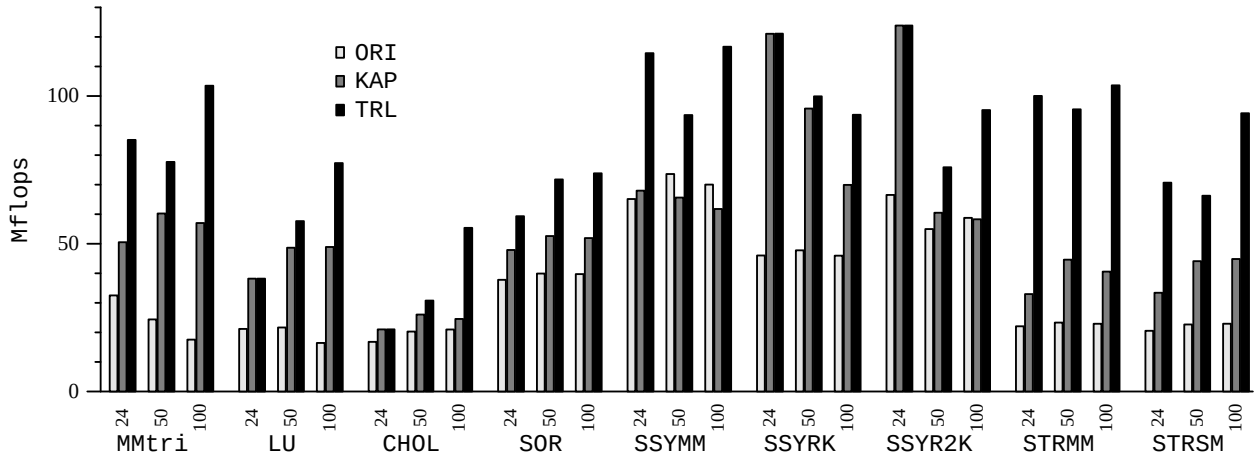

Figure 4. Performance obtained on the ALPHA processor by the ORI, KAP and TRL versions of each of the programs with matrix size 24, 50 and 100.

To evaluate the performance improvement of tiling for the register level, we show the MFLOP/s obtained for three different matrix sizes (24, 50 and 100 elements) on both processors and for the 9 benchmark programs when compiled using the different versions of the codes (Fig. 4 and 5). On the R10000 machine, only the ORI and TRL versions were used, since the native compiler by itself already uses the KAP preprocessor.

As it can be seen, tiling the register level is always better than the optimizations performed by the KAP and F77 compilers for all matrix sizes. Tiling the register level outperforms other compiler optimizations due to two reasons: 1) it always achieves ILP in the loop body [11] and 2) the number of load/store instructions is significantly reduced (register reuse). Transformations such as inner unrolling can also achieve good levels of ILP in most of the cases. However, when the inner loop carries a dependence it cannot increase the available ILP of the original code.

On the ALPHA processor, KAP performs better than the native F77 compiler in all programs except for SSYMM. In this case, the optimizations performed by KAP prevent optimizations that the F77 compiler would normally apply in the original code. The KAP preprocessor was able to perform unroll & jam (in only one dimension) for only four programs (SOR, SSYMM, SSYRK, SSYR2K). For the rest, it was only able to unroll (not fully) the innermost loop and apply scalar replacement. Moreover, the scalar replacement done by KAP was only used to move invariant references outside the innermost loop, and failed to reuse loads/ stores across successive iterations (this situation appears in program SOR, where our method did actually exploit this reuse).

In general, on the ALPHA processor, the performance improvement of TRL is much better for medium problem sizes (100) than for very small problem sizes (24) and, it is also much better for problem sizes multiple of the tile sizes (24 and 100) than for sizes not multiple of the tile sizes (50). For problem sizes that are very small and/or not multiple of the tile sizes, the execution time wasted on *boundary* tiles is significant and in these tiles less ILP and less data reuse than in *non-boundary* tiles is achieved [11]. To overcome this problem we have to consider the iteration space shape to evaluate temporal reuse, weighting each reuse direction with its corresponding ratio of *boundary* tiles versus total tiles executed

However, the same does not happen on the MIPS processor. The MIPS processor, due to its capability of issuing instructions out of order and speculating instructions beyond branches (four branches), unrolls loops dynamically. Therefore, the MIPS

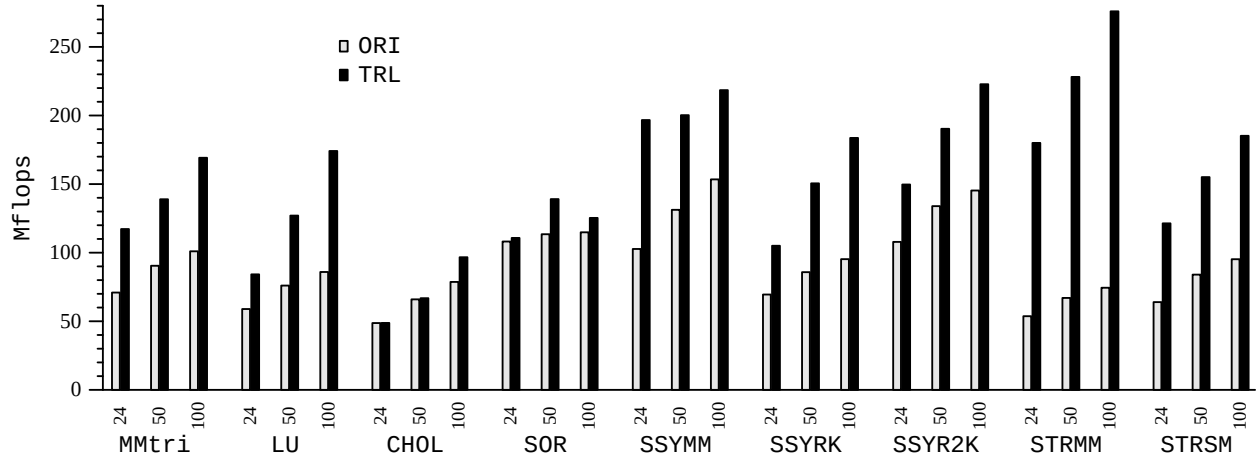

Figure 5. Performance obtained on the MIPS processor by the ORI and TRL versions of each of the programs with matrix size 24, 50 and 100.

processor is able to unroll the *element* loops of *boundary* tiles dynamically, extracting ILP from these tiles. In Fig. 5 it can be seen that the performance improvement of TRL with respect to ORI is almost constant for each matrix size.

In CHOL and LU programs, there is another reason of the little performance improvement achieved for small problem sizes. These programs perform very time consuming, non-pipelined operations (SQRT and DIV) and only when the matrix size increases, the execution time spent in these operations is hidden by the remaining operations.

Another point to notice is the behavior of the SSYRK and SSYR2K programs on the ALPHA processor. While the performance of all other programs increases with the problem size, the performance of SSYRK and SSYR2K decreases. The reason of this behavior is that, tiling for the register level can increase the TLB misses when spatial locality is not being exploited [11]. In the SSYRK and SSYR2K programs the *tile* loops selected to exploit temporal locality do not provide spatial locality. To solve this problem we could (a) perform tiling also for higher levels of the memory hierarchy or (b) consider also spatial reuse when the *tile* loops were selected.

Again, the same does not happen on the MIPS processor. The MIPS processor has double number of TLB entries and, thus, for these small problem sizes the TLB does not harm performance. Nevertheless, we note that for larger problem sizes the behavior of those programs on the MIPS processor is the same as on the ALPHA processor.

We also want to mention that TRL also achieves very good performance improvement for large problem sizes (1000, for example), although tiling for higher levels of the memory hierarchy has not been performed. A side effect of tiling the register level is a reduction of the overall cache misses because reducing the number of load/store instructions reduces data memory traffic. In Table 3 and Table 4 we have summarized the speedups of TRL over KAP (for the ALPHA processor) and over ORI (for the MIPS processor). For each program version the harmonic mean of the MFLOP/s obtained for different matrix sizes is computed. Then, we compute the speedup of TRL over KAP and ORI versions by dividing these harmonic means. Table 3 shows speedups for medium-small problem sizes (12 different problem sizes, going from 20 to 200) and Table 4 shows speedups for large problem sizes (21 different problem sizes, going from 500 to 1500). On the ALPHA, the speedups over the KAP preprocessor are in the range 1.15 to 2.23 for small problem sizes and between 1.56 and 5.96 for large sizes. On the MIPS, the speedups over the native compiler, vary between 1.11 and 3.28 for small problem sizes and between 1.28 and 5.06 for large sizes.

Finally, we want to indicate that our method increases code size and this fact could increase the instruction cache misses. However, we have seen by instrumenting the executables with the ATOM tool, that the overall instruction cache misses is insignificant for

Processor	MMtri	LU	CHOL	SOR	SSYMM	SSYRK	SSYR2K	STRMM	STRSM
ALPHA (TRL/ KAP)	1.54	1.29	1.27	1.35	1.63	1.15	1.44	2.23	1.87
MIPS (TRL/ ORI)	1.71	1.74	1.16	1.11	1.59	1.82	1.44	3.28	1.97

Table 3: Speedups obtained by TRL over the KAP and ORI versions on the ALPHA and MIPS processors, using small problem sizes (20-200).

Processor	MMtri	LU	CHOL	SOR	SSYMM	SSYRK	SSYR2K	STRMM	STRSM
ALPHA (TRL/ KAP)	1.67	2.33	5.96	1.88	1.97	1.56	2.03	2.32	2.12
MIPS (TRL/ ORI)	2.74	4.08	5.06	1.55	2.33	2.91	1.28	3.32	2.88

Table 4: Speedups obtained by TRL over the KAP and ORI versions on the ALPHA and MIPS processors, using large problem sizes (500-1500).

the 9 benchmark programs. The reason is that the generated code has a good degree of locality; in average for the 9 programs, 97% of the referenced instructions are done by only 10% of the executed code. Our method increases the static code size considerably because all different types of *boundary* tiles have to be considered, however, in execution time, only some of all loop nests in the code are actually executed (the loop nest that traverses the partition containing *non-boundary* tiles is the one that is executed the most).

5. RELATED WORK

There has been much discussion in the literature regarding memory hierarchy management [2] [5] [12][16] [20], but it has mostly focused on exploiting data reuse for the cache level and less attention has been paid to the register level.

M. Wolf in [19] presents a method to perform loop tiling on all levels of the memory hierarchy, but, at the register level, he only exploits data reuse in one dimension of the iteration space and indicates that tiling more loops at the register level is “not trivial”. Our method, however, is able to exploit data reuse at the register level in more than one dimension of the iteration space.

Carr in [2] and [3] uses unroll & jam for exploiting reuse at the register level and improving ILP. He handles limited cases of non-rectangular iteration spaces. In particular, he only allows one inner loop to have bounds that are affine function of only one iteration variable of tiled loops. Our work extends that of [2] and [3] by allowing several inner loops to have affine bounds of multiple tiled loops induction variables.

Moreover, for a set of different iteration space shapes, Carr gives the code transformation directly. To this end, he uses pattern recognition techniques on the bounds of the loops and when the iteration space shape does not match one of his patterns, no general algorithm to split these iteration spaces into simpler ones that could be recognized through patterns is presented. In some special cases, he uses index set splitting *before* applying unroll & jam to split the original iteration space into simpler ones. Our method, however, uses index set splitting *after* applying loop tiling. This fact makes unroll & jam and register tiling to generate different transformed codes for non-rectangular spaces. In particular, unroll & jam generates more *boundary* tiles than our proposed method.

Wolf, Maydan and Chen in [22] developed an algorithm that combines tiling for the cache level, unroll & jam and software pipelining to select a set of transformations leading to high performance. They handle non-perfectly nested loops and loops with non-rectangular iteration spaces, but they do not give details in [22]. To exploit the register level, they use unroll & jam as described by Carr and, as mentioned before, it can only be applied in limited cases of non-rectangular iteration spaces.

Finally, we note that current commercial preprocessors, such as KAP, are not able to perform tiling for the register level 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.

6. CONCLUSIONS

Tiling for the register level in more than one dimension has two main goals: (1) it exploits temporal data reuse that translates into a significant reduction of the number of load/store instructions issued (and, in most cases, this can lead to a reduction of the critical path length) and (2) it always improves the ILP of the original loop.

In this paper we have presented a new general method that performs tiling for the register level. The proposed method is distinguished from previous work primarily by being able to block in several dimensions of the iteration space in both rectangular and non-rectangular iteration spaces.

Our method applies first loop tiling to divide the iteration space into tiles. Afterwards it applies index set splitting repeatedly to distinguish loop nests that traverse *boundary* tiles of the tiled iteration space from loop nests that traverse *non-boundary* tiles. We have presented an algorithm to perform index set splitting repeatedly so that each loop in the nest will be processed only once and so that code expansion is reduced. We have also given analytical expressions to evaluate the complexity of our method and the amount of code generated.

Finally, we have evaluated the performance of our method applied to several linear algebra algorithms having non-rectangular iteration spaces. Our compilation technique has been compared against native compilers and against the commercial KAP preprocessor, on two different superscalar microprocessors. In all cases, our method has outperformed the native compilers and the KAP preprocessor, showing speedups in the range of 1.11 to 5.96.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] 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

[2] S. Carr. Memory-Hierarchy Management. Ph.D. Dissertation, Rice University, Feb 1993.

[3] S. Carr. Combining Optimization for Cache and Instruction-Level Parallelism.In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, Oct 1996, pp. 238-247.

[4] S. Carr, K. Kennedy. Compiler Blockability of Numerical Algorithms. International Conference on Supercomputing, 1992, pp. 114-124

[5] 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

[6] P. Havlak, K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. IEEE Transactions on Parallel and Distributed Systems. Vol. 2, No. 3, July 1991.

[7] F. Irigoin, R. Triolet. Supernode Partitioning. Symposium Principles Programming Languages, Jan 1988, pp.319-329

[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, Universidad Politécnica de Catalunya, Dec 1995.

[9] M. Jiménez, J.M. Llabería, A. Fernández, E. Morancho. Index Set Splitting to Exploit Data Locality at the Register Level. Technical Report UPC-DAC-1996-49, Dept. of Computer Architecture, Universidad Politécnica de Catalunya, Oct 1996.

[10] M. Jiménez, J.M. Llabería, A. Fernández. Loop Bounds Computation for Multilevel Tiling. 6th Euromicro Workshop on Parallel and Distributed Processing (PDP'98). Jan 1998, pp. 445-452.

[11] M. Jiménez, J.M. Llabería, A. Fernández. Performance Evaluation of Tiling for the Register Level. 4th Int. Symposium on High Performance Computer Architecture (HPCA-4). Feb 1998, pp. 254-267.

[12] K. Kennedy, K. M^{c}Kinley. Optimizing for Parallelism and Data Locality. International Conference on Supercomputing, July 1992, pp. 323-334

[13] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW machines. In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988, pp. 318-328

[14] 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

[15] 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.

[16] 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

[17] B. Rau. Iterative Modulo Scheduling. In Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27), Dec 1994, pp. 63-74

[18] R. Schreiber, J. Dongarra. Automatic Blocking of Nested Loops, Technical Report, RIACS, NASA Ames Research Center and Oak Ridge Nat'1 Laboratory, May 1990.

[19] M. Wolf. Improving Locality and Parallelism in Nested Loops. Technical Report CSL-TR-92-538, Stanford University, Aug 1992.

[20] M. Wolf, M. Lam. A Data Locality Optimizing Algorithm. International Conference on Programming Language Design and Implementation, June 1991, pp. 30-44.

[21] 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.

[22] M. Wolf, D. Maydan, D.K. Chen. Combining Loop Transformations Considering Caches and Scheduling. In Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), Dec 1996, pp.274-286

[23] M.Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, 1996.

[24] M. Wolfe. More Iteration Space Tiling. International Conference on Supercomputing, 1989, pp. 655-664.