MULTILEVEL TILING FOR NON-RECTANGULAR ITERATION SPACES

Marta Jiménez

Departamento de Arquitectura de Computadores
Universitat Politècnica de Catalunya
Barcelona (Spain). March, 1999

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor por la Universitat Politècnica de Catalunya

MULTILEVEL TILING FOR NON-RECTANGULAR ITERATION SPACES

Author: Marta Jiménez

Advisor: Agustín Fernández

A mis padres y,

especialmente, a Roger.

ABSTRACT

Microprocessor-based systems are increasingly becoming the workhorse for all scientific and engineering computation. They have numerical processing capabilities that already rival older generations of supercomputers. Over the last decade, microprocessor design strategies have focused on increasing the computational power available on a single chip. These advances in computational capacity have been achieved by reducing cycle time and also via architectural changes such as pipelined floating-point functional units, multiple instruction issue and out-of-order execution. Nevertheless, a high computation bandwidth is meaningless unless it is matched by a similarly powerful memory subsystem. Unfortunately, while on-chip operation speeds have improved dramatically, the performance of memory has not. The result has been an imbalance between computation speed and memory speed and this imbalance has led machine designers to use complex memory systems based on a hierarchy of levels.

However, to achieve high levels of performance it is not enough by improving the hardware structures. Compiler techniques should also be developed to fully utilize the architectural advances. In fact, the efficiency of architectural improvements depends on the compiler ability to change the structure of programs for taking full advantage of them. In the past few years, compilers have provided some support both to improve ILP and to optimize code for complex memory hierarchies. However, existing compiler technology is oriented mostly towards simple numerical codes containing simple loop nests that describe rectangular iteration spaces. This is understandable since transformations are easy to apply on this type of loop nests. Nevertheless, several linear algebra algorithms typically used in numerical codes consist of complex loop nests defining non-rectangular iteration spaces. Current commercial compilers are unable to restructure and optimize these types of codes and, therefore, poor performance is achieved on these complex loop nests.

This fact has led many programmers to restructure their algorithms by hand to perform well on particular memory systems, a situation that has lead to machine-specific programs. Additionally, manufacturers have tried to minimize the complexity of writing optimized code by providing numerical libraries that attain high performance under their machine. However, not all applications can take advantage of these libraries and there are many situations in which none of the routines provided can specifically solve the task at hand. We believe that restructuring a code to achieve high performance should be the job of the compiler. Compilers, not programmers, should handle the machine-specific details required to attain high performance on each particular architecture.

The main motivation of this thesis is to develop new compilation techniques that address the lack of performance of complex numerical codes consisting of loop nests defining non-rectangular iteration spaces. Specifically, we focus on the loop tiling transformation (also known as blocking) and our purpose is the improvement of the loop tiling transformation when dealing with complex numerical codes. Our goal is to achieve via the loop tiling transformation the same or better performance as hand-optimized vendor-supplied numerical libraries.

We will observe that the main reason why current commercial compilers perform poorly when dealing with this type of codes is that they do not apply tiling for the register level. Instead, to enhance locality at this level and to improve ILP, they use/combine other transformations that do not exploit the register level as well as loop tiling. Tiling for the register level has not generally been considered because, in complex numerical codes, it is far from being trivial due to the irregular nature of the iteration space. Our first contribution in this thesis will be a general compiler algorithm to perform tiling at the register level that handles arbitrary iteration space shapes and not only simple rectangular shapes.

Our method includes a very simple heuristic to make the tile decisions for the register level. At first sight, register tiling should be performed so that whichever loop carries the most temporal reuse is not tiled. This way, register reuse is maximized and the number of load/store instructions executed is minimized. However, we will show that, for complex loop nests, if we only consider reuse directions and do not take into account the iteration space shape, the tiled loop nest can suffer performance degradation. Our second contribution will be a proposal of a very simple heuristic to determine the tiling parameters for the register level, that considers not only temporal reuse, but also the iteration space shape. Moreover, the heuristic is simple enough to be suitable for automatic implementation by compilers.

However, to be able to achieve similar performance to hand-optimized codes, it is not enough by tiling only for the register level. With today's architectures having complex memory hierarchies and multiple processors, it is quite common that the compiler has to perform tiling at four or more levels (parallelism, L2-cache, L1-cache and registers) in order to achieve high performance. Therefore, in today's architectures it is crucial to have an efficient algorithm that can perform multilevel tiling at multiple levels of the memory hierarchy. Moreover, as we will see in this thesis, multilevel tiling should always include the register level, as this is the memory hierarchy level that yields most performance when properly tiled.

When multilevel tiling includes the register level, it is critical to compute exact loop bounds and to avoid the generation of redundant bounds. The reason is that the complexity and the amount of code generated by our register tiling technique both depend polynomially on the number of loop bounds. However, to date, the drawback of generating exact loop bounds and eliminating redundant bounds has been that all techniques known were extremely expensive in terms of compilation time and, thus, difficult to integrate in a production compiler. Our third contribution in this thesis will be a new implementation of multilevel tiling that computes exact loop bounds at a much lower complexity than traditional techniques. In fact, we will show that the complexity of our implementation is proportional to the complexity of performing a loop permutation in the original loop nest (before tiling), while traditional techniques have much larger complexities. Moreover, our implementation generates less redundant bounds in the multilevel tiled code and allows removing the remaining redundant bounds at a lower cost. Overall, the efficiency of our implementation makes it possible to integrate multilevel tiling including the register level in a production compiler without having to worry about compilation time.

The last part of this thesis is dedicated to studying the performance of multilevel tiling. We will discuss the effects of tiling for different memory levels and present quantitative data comparing the benefits of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. Finally, we will compare automatically-optimized codes against hand-optimized vendor-supplied numerical libraries, on two different architectures (ALPHA 21164 and MIPS R10000), to conclude that compiler technology can make it possible for complex numerical codes to achieve the same performance as hand-optimized codes on modern microprocessors.

AGRADECIMIENTOS

Deseo expresar mi más sincero agradecimiento a todas las personas que, con su ayuda, han hecho posible la realización de este trabajo:

A Agustín Fernández, por dirigir esta tesis y animarme continuamente a seguir adelante.

A José M. Llabería, por supervisar todo este trabajo, por todo lo que me ha enseñado y por todo el tiempo que me ha dedicado.

A Dolors y a Toni, por acompañarme durante todos estos años, por su ayuda y sus ánimos.

A Sergi y a Mildred, por aceptarme durante tantos años como compañera de despacho.

A todo el personal del LCAC y a los administradores de sistemas del CEPBA, por su excelente soporte técnico.

A Anna, Luis, Montse, Enric, Jose Ramón, Pepe y a todo el departamento de Arquitectura de Computadores, por ser magníficos compañeros de trabajo.

A Herr Huther, Herr Boyer y, muy especialmente, a Don José Cruz Santana, por todo lo que me enseñaron durante mi infancia.

A Pau Calpe y Gonzalo Martín, por estar a mi lado en los momentos más difíciles y por todo lo que hemos compartido y vivido juntos.

A mis tíos, Ramón y Ma. Angela, y a mis primas, Carmen, Elisabet y Angela, por acogerme durante mis primeros años en Barcelona y hacer que me sintiera como en casa.

A toda mi familia de Barcelona, especialmente a Ramón Espasa, Carme Sans, Marina Espasa y Carme Alfonso, por todo su cariño y por su apoyo continuo durante todos estos años.

A mi hermana Carmen, por hacerme reir cada día con sus divertidísimos e-mails y por ser algo más que una hermana.

A mis padres, José Luis y Carmen, y a mis hermanos, José Luis, Ching Lee, Isaac, Cristian y Carmen, por su infinito amor, por su incesante apoyo y por mucho más de lo que puedo expresar con palabras.

Finalmente, y con todo mi cariño, a Roger. Sin ti, este trabajo no habría sido posible.

This work was supported in part by the Ministry of Education of Spain under contracts TIC-92-0880, TIC-95-0429 and TIC-98-0511 and by the CEPBA (European Center for Parallelism of Barcelona).

CONTENTS

1 INTRODUCTION

Summary

In this chapter we present the motivation behind this thesis. Our goal is to improve current compiler technologies to achieve high performance on complex numerical codes. The main reason numerical codes perform poorly on modern microprocessors is that they do not utilize the memory system effectively, specially the register level. In this chapter, we review different hardware and compiler techniques developed to maximize the effectiveness of the memory system and we explain why current commercial compilers still perform poorly when dealing with complex codes. Finally, we conclude with a thesis overview, a summary of related work and the thesis organization.

1.1 MOTIVATION

Microprocessor-based systems are increasingly becoming the workhorse for all scientific and engineering computation. They have numerical processing capabilities that already rival older generations of supercomputers and the microprocessors used in these systems will continue to improve due to every-increasing clock rates and new architectural advances. In contrast to the vector-based machines that have long dominated high performance computing, these new scalar systems are considerably more cost-effective since they contain commercial microprocessors that are mass-produced for the large general-purpose computing market. In addition, these microprocessors can be used to build large-scale multiprocessors capable of aggregate peak rates sometimes improving that of current vector machines.

The performance of a microprocessor-based computer is determined by the performance of its memory system. While on-chip operation speeds have improved dramatically, the performance of memory has not. Thus as the processor cycle time comes down, the latency to main memory has been increasing. To ameliorate these problems, machine designers have turned to complex memory systems based on a hierarchy of levels. The idea behind a *Memory Hierarchy* is to place a small high-speed memory close to the processor which is backed up by increasingly larger but slower memories. The key issue for high performance under a memory hierarchy is the minimization of data transfers between the different memory levels.

Unfortunately, modern microprocessors using such memory hierarchies perform poorly on numerical applications. Numerical codes tend to operate on very large data sets that do not fit in the small high-speed memories that are close to the processor. The result is a large amount of data being transferred between the different memory levels, yielding poor memory system performance. However, numerical codes tend to have patterns of data usage that are regular in structure and that include opportunities for data reuse. These regularities have led many programmers to restructure their algorithms by hand to increase data reuse and better exploit the memory hierarchy. The drawback of this approach is that manually optimized codes are difficult to write, difficult to debug and, even worse, hardly understandable once finished. Moreover, manually optimizing a certain code introduces machine parameters that have nothing to do with the problem being solved and which must be adjusted for each computer where the algorithm must be run if good performance is to be achieved.

Manufacturers have tried to minimize the complexity of writing optimized code by providing numerical libraries that attain high performance under their particular memory hierarchy. The BLAS3 library [38][39], for example, provides a set of standard linear algebra operations which are highly optimized for each specific machine. On top of the BLAS standard interface, higher level library packages such as LAPACK[1][9] have been built. These higher level packages provide a rich variety of mathematical algorithms that take advantage of the BLAS3 vendor-optimized routines. However, not all applications can take advantage of these libraries and there are many situations in which none of the routines provided can specifically solve the task at hand.

We believe that restructuring a code to better exploit the memory hierarchy should be the job of the compiler. Although the library approach just described can handle some situations, in general we believe that compilers should take the responsibility of optimizing code to exploit the memory hierarchy. Compilers, not programmers, should handle the machine-specific details required to attain high performance on each particular architecture. Algorithms should be expressed in a natural, machine-independent form and the compiler should apply the appropriate transformations to optimize the resulting code.

In the past few years, compilers have provided some support to optimize code for complex memory hierarchies. Compiler algorithms to manage the memory hierarchy automatically have been developed and shown through experimentation to be effective. However, existing compiler technology is oriented mostly towards simple numerical codes containing simple loop nests with constant loop bounds. This is understandable since transformations are easy to apply on this type of loop nests. Nevertheless, several linear algebra algorithms typically used in numerical codes consist of complex loop nests that have complex functions as their loop bounds. Current commercial compilers are unable to restructure and optimize these types of codes and, therefore, poor performance is achieved on these complex loop nests.

As an example, Fig. 1.1 shows the performance (in Mflop/s) obtained by the linear algebra problem STRMM, varying the problem size. Measurements were taken on a R10000 processor. The STRMM problem is a matrix by matrix multiplication with one of the matrices being triangular. The circle curve shows the performance obtained if we directly compile the code of STRMM using the f77

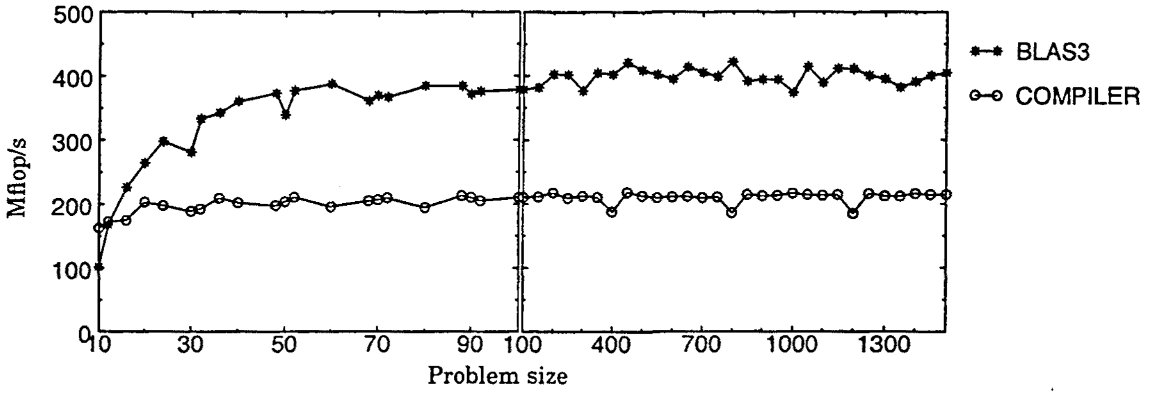


Figure 1.1: Performance of STRMM on the R10000 processor, varying the problem size.

1.LAPACK, "Linear Algebra Package", is a project originated by Jack Dongarra. This project put together a new set of linear algebra functions, supposed to supplant both the LINPACK and EISPACK packages. To achieve maximum efficiency across all types of hardware, the LAPACK routines are based on the BLAS3 routines.

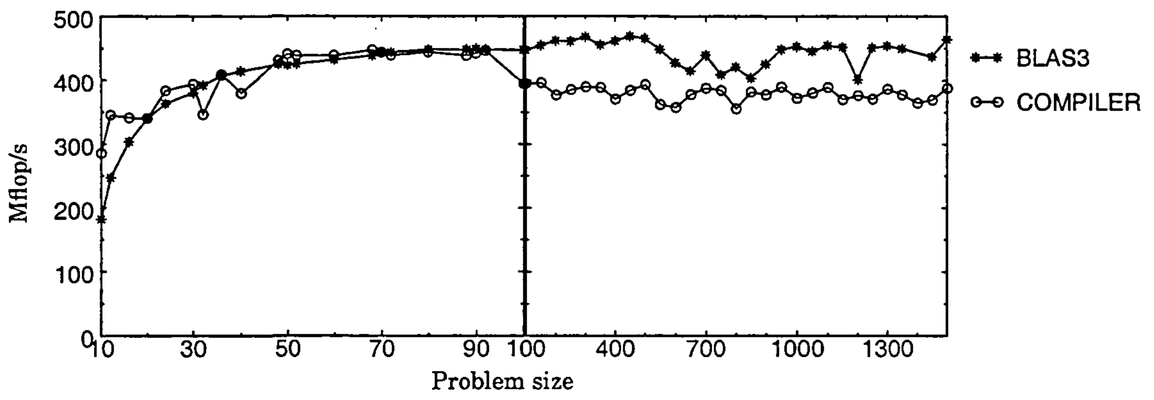


Figure 1.2: Performance of SGEMM on the R10000 processor, varying the problem size.

compiler with maximum level of optimization. The star curve shows the performance obtained if we call the vendor-optimized BLAS3 library [38] to perform the operation. It is well known that the BLAS3 library is highly hand-optimized to properly exploit the machine characteristics. We can see how current compilers achieve poor performance compared with the hand-optimized code provided by the BLAS3 library.

In contrast, let's now look at how commercial compilers behave on *simple* numerical codes. Figure 1.2 shows the performance obtained by the linear algebra problem SGEMM. SGEMM consists of a very simple loop nest, performing a rectangular matrix by matrix multiplication. We can see how, in this case, the native compiler is able to achieve higher performance than in the STRMM problem. Nevertheless, the hand-optimized BLAS3 library still performs better for large problem sizes.

The conclusion is that, despite all the effort put in current compilers to achieve high performance in numerical codes, hand-optimized codes still outperform them. Moreover, this performance difference between hand-optimized codes and automatic-optimized codes is more noteworthy in complex numerical codes as seen in the STRMM problem.

The goal of this thesis is to develop new compilation techniques that address the lack of performance of complex numerical codes when run under a memory hierarchy. Compiler algorithms to restructure complex numerical codes are developed and shown through experimentation to be effective.

The remainder of this chapter reviews different hardware techniques developed for coping with the memory latency problem and different compilers strategies proposed for fully realizing the architectural advances. Then we will explain why current commercial compilers still perform poorly when dealing with complex scientific applications. Finally, we conclude this chapter with a thesis overview, a summary of related works and the thesis organization.

1.2 THE MEMORY LATENCY PROBLEM

Over the last decade, microprocessor design strategies have focused on increasing the computational power available on a single chip. This advances in power have been achieved by reducing cycle time and also via architectural changes such as pipelined floating-point functional units, multiple instruction issue and out-of-order execution.

Unfortunately, a high computation bandwidth is meaningless unless it is matched by a similarly powerful memory subsystem. Although microprocessor speed has been increasing dramatically the speed of memory has not kept pace. Figure 1.3 shows the evolution, in relative terms, of microprocessors and DRAM speeds over the past 16 years. As it can be seen, from 1986 onward the microprocessor cycle time has been improving at a rate of 60% per year, approximately. Meanwhile, DRAM speeds have been slowly improving at a rate of only 7% per year. Producers of DRAM's have mostly focused on improving cost per bit and density per chip rather than favoring faster cycle times. The result is that, although density has gone from 16 Kbits in a chip in 1976 to 64 Mbits in a chip in 1996 (a 4000-fold improvement), these chips have only moved from a 400ns cycle time down to a 80ns cycle time (a 5-fold improvement). Today, memory chips are on the order of 10 to 100 times slower than CPUs.

Also, the instruction level parallelism available in recent microprocessors has increased. Since several instructions are being issued in the same processor cycle, the total amount of data requested per cycle to the memory system is much higher. This increase in requested data can be partially supported by an adequate increase in the number of memory ports. However, while most of current microprocessors are able to issue up to 4 instructions per cycle, the number of memory ports has not increased enough to support this degree of instruction parallelism.

These factors have led to a situation where each off-chip memory access can have extremely large latencies. As a result, and depending on the degree of cache-friendliness, the total execution time of a program can be greatly dominated by average memory access time.

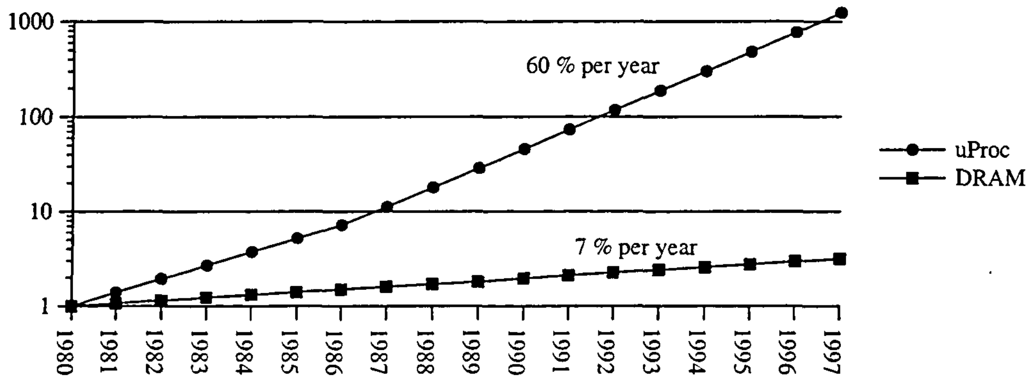


Figure 1.3: Relative evolution of microprocessor and DRAM speeds (source [101]).

1.3 COPING WITH MEMORY LATENCY

The memory latency problem has been attacked from two different fronts. First, computer architects have proposed several hardware mechanisms that ameliorate the memory latency problem: lockup-free caches, memory hierarchies, prefetching, out-of-order execution, etc. Second, compiler techniques have been developed to fully utilize the hardware structures available. In this section we will briefly review different hardware techniques developed for coping with the memory latency problem and different compilers strategies proposed for fully realizing these hardware proposals.

1.3.1 Hardware Strategies

To reduce the memory latency problem, computer architects have proposed several strategies that can be divided into two categories: those for tolerating memory latency and those for avoiding latency. *Latency tolerance* means doing "something else" while data is being fetched from memory while *latency avoidance* tries to approximate memory access time to processor cycle time. The result of both techniques is a reduction on the number of processor cycles spent waiting for memory accesses to complete. Hardware strategies for tolerating latencies include write buffers, non-blocking loads, lockup-free caches, prefetching, out-of-order execution and multithreading, and the most common hardware mechanisms for latency avoidance are memory hierarchies, victim caches and pseudo-associative caches.

One way to tolerate memory latency is to allow memory references to be buffered. This technique was initially applied only to writes in the form of *write buffers*. Using *write buffers*, the processor does not have to wait for a write to complete. Instead, it performs a write by simply issuing it to the write buffer, a simple operation that is performed in one cycle. The advantage of a *write buffer* is not only that the processor does not stall when executing a write, but also that bus contention is reduced by delaying writes until idle bus cycles occur [61].

Buffering read accessed is more difficult because, unlike writes, the processor typically cannot proceed until the read access completes, since it needs the data that is being read. With *non-blocking loads* and *lockup-free caches* it is possible to buffer read accesses. A *non-blocking load* [43] allows the CPU to continue executing on a cache miss. That is, rather than stalling at the time a load is performed, the processor postpones stalling until the data is actually used, thus hiding the latency of the memory request with the following independent instructions. *Non-blocking loads* were later combined with *lockup-free caches* [75] to escalate their potential benefits. A *lockup-free cache* is able to supply data resulting from a cache hit even while processing a prior miss. Also, *lockup-free caches* typically allow multiple outstanding cache misses, thus giving the processor more possibilities to continue executing instructions while the misses are being served.

Another hardware advance to tolerate memory latencies is *hardware prefetching* [12][47][100][114]. It tolerates memory latency by performing data requests sufficiently far in advance of the use of the data in the execution stream. Obviously, this approach requires lockup-free caches so that processors can proceed while prefetched data are being fetched. Prefetching also requires the ability to predict which data items are needed ahead of time, a difficult task to be done in hardware and that can yield to a high rate of useless prefetching that can actually lower performance. An alternative to *hardware prefetching* is for the compiler to insert prefetch instructions to request the data before they are needed [72][93][102][119]. Issuing prefetch instructions incurs an instruction overhead, thus, care must be taken to ensure such overhead does not exceed the benefits.

Besides these hardware mechanisms, in the last decade there has also been important architectural improvements, such as *out-of-order* and *multithreaded* execution, to attack the memory latency problem. *Out-of-order* architectures [8][68][117][130] attack the memory latency problem by allowing memory accessing instructions to precede while other instructions are waiting for memory data. That is, memory instructions are allowed to slip ahead of execution instructions. *Multithreaded* processors [2][41][118], however, deal with the memory latency problem by switching between threads of execution every time a long latency operation (such as a cache miss) threatens to halt the processor.

All previous strategies cope with the memory problem by tolerating latencies. Another way of dealing with this problem consists of avoiding latencies. *Memory hierarchies* [53] are the most widespread approach for latency avoidance. It consists of placing small high-speed memories close to the processor and larger slower memory further from the processor. If data that will be used multiple times are stored in the fast memory closer to the processor, then they can be retrieved more quickly in following requests, thus reducing the average memory latency.

The likelihood of finding data in the faster memory levels depends not only on the size, replacement policies and organization of the levels, but also on the inherent locality of reference within the applications. The principle of locality says that if a program refers to any given word in memory, there is a high likelihood that in the near future the program will refer to the same location in memory (temporal locality) or a nearby location (spatial locality). Since most applications exhibit a reasonable amount of locality, memory hierarchies are generally quite useful. The whole hierarchy in a modern virtual memory system may include up to six different memory levels: disk-backing store, main memory, one, two or even three levels of high speed cache memory and the register file[2].

More recent approaches to avoid memory latency are *victim caches* and *pseudo-associative caches*. Victim caches [62] improve the performance of direct-mapped caches through the addition of a small, fully associative cache between the first level cache and the next level in the hierarchy. Pseudo-associative caches [3][19][64][112], however, consist in cache organizations that modify a

2. Registers are often considered separate from the rest of the memory hierarchy, because they are managed explicitly by the compiler; however we think, and so we do in this work, that registers can be fruitfully treated as part of the hierarchy.

set-associative cache to achieve an average access time close to that of a direct-mapped cache, thus obtaining the miss rate of set-associative caches and the hit speed of direct-mapped. Basically, victim and pseudo-associative caches both promise to improve miss rate without affecting the processor clock rate.

1.3.2 Compiler Strategies

Due to design and technology limitations, the efficiency of all previous hardware mechanisms and architectural improvements depends on the compiler ability to change the structure of programs for taking full advantage of them. For instance, mechanisms that tolerate memory latencies are effective if the compiler can schedule instruction streams with enough parallelism between the instructions, while latency avoidance mechanisms are effective only when a compiler can determine that values will be reused and should be kept in the fastest level of the memory hierarchy. Thus, to fully realize all recent architectural advances and, therefore, achieve high performance on modern superscalar processors, compilers need to find ILP to utilize machine resources effectively, and they also need to transform the program to achieve a high degree of data locality to maximize the effectiveness of the memory system.

When optimizing a program, the most gains will come from optimizing the regions of the program that require the most time. These regions usually are the repetitive regions that correspond to iterative loops whose loop body contains n-dimensional array variables. In this work we focus on this type of regions.

Several compiler strategies have been developed to exploit a program's ILP and/or to improve the memory hierarchy utilization inside loop nests. These strategies can be classified into two different classes: strategies that change the original data layout of the array variables and strategies based on *loop restructuring transformations* that reduce the number of executed instructions and/or change the order in which statements are executed.

Strategies that change the original data layout of array variables are *padding* (or *data alignment*) and *data transformations*. Padding [11][82][107] is a technique that involves the insertion of dummy elements in a data structure, while data transformations [10][63][66][92][99] consist on simply reorganizing the original data layout. The goal of both techniques is avoiding self and cross interferences, thus reducing conflict misses and improving the memory hierarchy utilization. These techniques are usually combined with loop restructuring transformations to yield high performance.

Next, we review the most relevant loop restructuring transformations that aim at improving ILP and/or the memory hierarchy utilization [24][79][124]; we note that most of these transformations can be combined together to maximize the effectiveness of the memory system and, at the same time, improve ILP.

Compiler transformations such as *inner unrolling*[3] [129] and *software pipelining* [56][79][106] were proposed to improve ILP, but they do not aim at enhancing data locality. Both techniques draw out parallelism between iterations of the original loop body, but they are implemented in different ways. *Inner unrolling* replaces the body of the loop by several copies of the body and adjusts the loop-control code accordingly. *Software pipelining* reorganizes loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. Both transformations, in addition to yielding a better scheduled inner loop with a good degree of parallelism, each reduce a different type of overhead. *Inner unrolling* reduces the overhead of the loop (the branch and counter-update code) and *software pipelining* reduces the number of times that the loop is not running at peak speed to once per loop at the beginning and end. Because these techniques attack two different types of overhead, the best performance is usually obtained by combining them.

There are other compiler transformations whose goal is to enhance data locality at the cache level, but not to improve program's ILP. These transformations are *loop permutation*, *loop fusion* and *loop fission* [96][129]. Some programs have nested loops that access data in memory in non-sequential allocated order. Simply *permuting* the loops in the nest can make the code access the data in the order it is stored, thus improving spatial locality. *Loop fusion* enhances temporal locality by combining loops when they use the same data [88][113]. Some programs have separate sections of code that access the same arrays, performing different computations on the common array. By fusing the code into a single loop, the data that are loaded into the cache can be used repeatedly before being swapped out. Finally, *loop fission* (the inverse of *loop fusion*) can also improve memory locality by breaking up a loop that refers to a large amount of data into a sequence of smaller loops, each of which has disjoint or smaller data requirements. This way, the program's working set is reduced ad therefore data locality improved.

Scalar replacement [20][23] is a code transformation that also enhances data locality, but at the register level. One concern in generating good object code for loops that manipulate subscripted variables is that very few compilers even try to allocate such variables to registers, despite the fact that register allocation is often done very effectively for non-subscripted scalar variables. *Scalar replacement* finds opportunities to reuse array elements and replaces the reuses with references to scalar temporaries, hence making them available for register allocation.

A compiler transformation which is able both to improve ILP and to enhance data locality is *outer unrolling* (also called *unroll-and-jam*) [20][23]. *Outer unrolling* draws out parallelism between iterations of the original loop body and enhances data locality at the register level in the unrolled dimension of the iteration space. It consists in unrolling an outer loop in a nest and then jamming the resulting inner loops back together. *Outer unrolling* can introduce more computation into an innermost loop body without a proportional increase in memory references. We note that *outer*

3. We use the term *inner unrolling* to refer to *unrolling* to clearly distinguish it from *outer unrolling*.

unrolling can be applied repeatedly to several loops in a nest and, in this case, data reuse is exploited in all the unrolled dimensions. It is well known that exploiting data reuse in several dimensions of the iteration space, whenever possible, improves the performance of the memory hierarchy [80][97].

Finally, another compiler transformation, and perhaps the most famous one, to enhance data locality is *Loop Tiling* (also called *Blocking*) [97] [110][121][128]. *Loop tiling* has been typically used to enhance data locality at the cache level. It consists in dividing the iteration space defined by the loop structures into regular tiles, creating a blocking of the data arrays. The order in which the tiles are traversed determines 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 the probability of finding the associated data in the memory level being exploited is higher.

Futhermore, *loop tiling* has other interesting properties. First, it exploits data reuse in several dimensions of the iteration space [98][121]. As mentioned before, this capability can improve the effectiveness of the memory level being exploited. Second, by performing *multilevel tiling*, data locality can be enhanced in several memory levels simultaneously. *Multilevel tiling* consists in repeatedly applying *loop tiling* for each level, dividing a tile of a higher level into subtiles. Each level of tiling exploits one level of the memory hierarchy. And third, *loop tiling* for the register level has the desirable property that it always increases ILP.

Thus, *loop tiling* is a very powerful transformation that has the potential of outperforming the other code transformations mentioned above since it is capable to achieve the three main optimizations mentioned so far:

- improves a program's ILP, when it is applied at the register level,
- exploits data reuse in *several* dimensions of the iteration space and
- enhances data locality at several memory levels simultaneously.

We want to note that loop tiling is implemented by combining other loop transformations, such as loop permutation, unrolling and scalar replacement. Although loop tiling can be seen as a combination of other transformations, we refer to it as a single transformation because it is a particular combination that yields to blocked algorithms.

As a quick summary, Table 1.1 presents all previous mentioned compiler transformations specifying for each one its main goals (improves ILP and/or data locality), at which level data locality is enhanced and in how many dimensions data reuse is exploited. We want to note that most of the transformations may indirectly achieve other benefits; for instance, scalar replacement might improve ILP by reducing the number of memory instructions executed in the loop body. In Table 1.1, however, we only indicate the main goals of each transformation, ignoring secondary effects that the transformation might have.

Compiler Transformation	**Improves ILP**	**Improves Data Locality**	**Number of reuse dimensions exploited**
inner unrolling	Yes	No	-----
software pipelining	Yes	No	-----
loop permutation	No	at cache level	1
loop fusion	No	at cache level	1
loop fission	No	at cache level	1
scalar replacement	No	at register level	1
outer unrolling	Yes	at register level	several*
loop tiling	Yes	at several memory level simultaneously	several

Table 1.1: Summary of the goals of different compiler transformations. (*) Outer unrolling can be applied to more than one loop in a nest and, in this case, it exploits data reuse in all the unrolled dimensions.

There are several works that focus on combining all these transformation to achieve high performance [24][27][108][122][124]. They try to select the best combination of transformations that yield better performance. The set of transformations that they manipulate to select a good combination is limited by its legality[4] and also by the ability of being able to generate the transformed code.

In this thesis we will focus on the loop tiling transformation, since it is the transformation that individually achieves more performance. The other loop transformations, if taken individually, realize some of the goals that loop tiling achieves, but not as many. Therefore, the main motivation of this thesis will be the improvement of the loop tiling transformation. Of course, loop tiling can sometimes be combined with some of the previous transformations to further improve its performance. For instance, loop fusion can be applied before loop tiling to enhance temporal locality. However, this thesis will look at the performance of the loop tiling transformation in isolation.

1.4 LOOP TILING IN COMMERCIAL COMPILERS

With today's architectures having complex memory hierarchies, it is necessary that the compiler performs tiling at three or more levels (L2-cache, L1-cache and registers) to achieve high performance. Previous work has shown that, when tiling for multiple memory levels, the register level is the most important one, more so than the cache levels [23][58][80], although for loop nests with very large working sets, tiling for various cache levels is also important. Being able to enhance data locality at

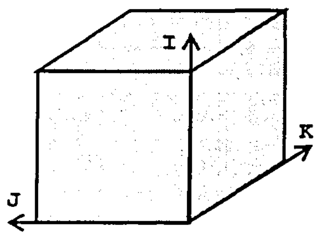
4. A loop transformation is *legal* if it does not violate the original data dependences.

the register level is extremely important in today's superscalar microprocessors, since the register level directly feeds the processor functional units. The typical bandwidth provided between the register file and the functional units is three or four times larger than the bandwidth provided by the first level cache. Typically, one can find at least 6 ports in the floating point register file whereas only one or two ports in the first level cache are provided. If the register level is not properly exploited, then the number of first level cache ports bounds processor performance. In general, when performing multilevel tiling the compiler should always include the register level in order to achieve high performance.

How do current commercial compilers deal with these issues? First of all, let's define a *simple* loop nest as a set of nested loops whose bounds are constant and, therefore, describe a *rectangular* iteration space. On the other hand, we define a *complex* loop nest as a set of nested loops whose bounds are maximum or minimum of affine functions of the surrounding loops iteration variable. This kind of loop nests describe *non-rectangular* iteration spaces. As an example, Fig. 1.4 shows the code and its associated iteration space for both a simple and a complex loop nest. The codes correspond to the SGEMM and the STRMM problems used in Section 1.1.

We have observed that current commercial compilers perform quite well when dealing with simple loop nests. For these codes, compilers essentially perform loop tiling for several memory levels. For small problem sizes they perform tiling only for the register level and, for large problem sizes, where cache levels affects processor performance, they also perform tiling for these other levels. Only when loop tiling is not a *legal* transformation, compilers use combinations of other transformations, such as inner and outer unrolling, software pipelining, loop fusion and loop fission, that are less restrictive from the legality point of view.

(a)
```
do J = 1, N
  do K = 1, N
    do I = 1, N
      C(I,J) = C(I,J) + D(K,J) * A(I,K)
  . . .
enddo
```



(b)
```
do J = 1, N
  do K = 1, N
    do I = 1, K-1
      D(I,J) = D(I,J) + D(K,J) * A(I,K)
  . . .
enddo
```

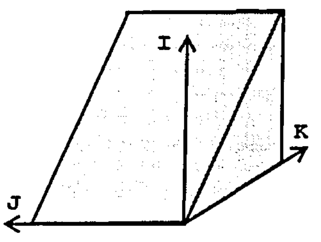


Figure 1.4: a) Simple loop nest code and its iteration space shape (SGEMM) b) Complex loop nest code and its iteration space shape (STRMM).

For complex loop nests, however, current compilers perform poorly. For these codes, compilers only perform loop tiling at the cache level. They do not apply tiling to the register level, despite it is legal. Instead, to enhance locality at the this level, they use or combine other transformations, that do not exploit the register level as well as loop tiling.

Now, our question is why compilers do not apply loop tiling at the register level in complex loop nests despite being legal? Tiling for the register level is not generally considered because the transformed loop nest is not easy to rewrite. Generating a transformed tiled loop nest for the register level consists of two steps: rewriting the body of the loop nest and rewriting the loop bounds.

At the register level, after dividing the original iteration space into tiles, it is necessary to rewrite the loop body by fully unrolling the loops that traverse the iterations inside the register tiles, because registers are only addressable using absolute addresses. In complex loop nests, the action of fully unrolling the loops is far from being trivial due to the irregular nature of the iteration space[121]. Currently, production compilers can only perform this unrolling for simple loop nests. In this thesis, we propose an implementation of tiling for the register level that handles arbitrary iteration space shapes and not only rectangular shapes.

Rewriting the loop bounds is also more difficult in complex loop nests than in simple loop nests. The bounds of a tiled loop nest can be *exact* or not. We say that a loop nest has *exact* bounds if it never executes an empty iteration. As an example, the code of Fig. 1.5a has exact loop bounds while the code of Fig. 1.5b has not. Every time K is equal to N, loop I does not perform any iteration. In addition, a loop nest can have *redundant* bounds. We say that a loop bound is *redundant* if it can be removed from the loop and the resulting loop nest executes exactly the same iterations as the original loop nest. In Fig. 1.5c, the lower bound J of loop I is redundant.

Computing *exact* bounds and avoiding the generation of *redundant* bounds is critical when multilevel tiling includes the register level. As mentioned before, when tiling is being applied at the register level, it is necessary (after dividing the original iteration space into tiles) to fully unroll the loops that traverse the iterations inside the register tiles. The complexity of these second step of register tiling and the amount of code generated depend on the number of bounds of the loops that have to be fully unrolled, and thus, it is convenient to compute *exact* and *non-redundant* bounds when multilevel tiling includes the register level.

```
do J = 1, N
  do K = 1, J
    do I = K, N
...
enddo
```
(a)

```
do J = 1, N
  do K = J, N
    do I = K+1, N
...
enddo
```
(b)

```
do J = 1, N
  do K = J, N
    do I = max(J, K), N
...
enddo
```
(c)

Figure 1.5: a) Loop nest with *exact* bounds b) Loop nest without *exact* bounds. Every time K is equal to N, loop I does not perform any iteration. c) Loop nest with *redundant* bounds. The lower bound J of loop I is redundant.

Moreover, another advantage of computing *exact* and *non-redundant* bounds is to avoid increasing a program's execution time. If the compiler does not compute *exact* bounds and generates *redundant* bounds, a fraction of a program's execution time is wasted in evaluating useless bounds (*redundant* bounds or bounds of loops that will end up in empty iterations). This fraction of time is usually insignificant, but it can become important if loop tiling is applied to several levels of the memory hierarchy.

Traditionally, exact loop bounds computation has not been performed because its complexity is doubly exponential and, therefore, for certain classes of loops, can be extremely time consuming. Of course, for simple loop nests that define rectangular iteration spaces, the cost of computing exact bounds is "reasonably" cheap. However, this is not the case for complex loop nests defining non-rectangular iteration spaces. In this thesis, we also propose an implementation of multilevel tiling that computes *exact* bounds, tries to avoid *redundant* bounds and its cost is sufficiently low that is viable to be implemented in a production compiler.

1.5 THESIS OVERVIEW

This thesis is focused on the study of a code transformation to enhance data locality and to extract ILP from numerical codes. More precisely, we center on the loop tiling transformation applied to loop nests that define *non-rectangular* iteration spaces. For this kind of loop nests, that commonly appear in numerical applications, current compilers are not able to perform multilevel tiling and, thus, high performance is not achieved. In this thesis we show that multilevel tiling can also be applied to loops defining *non-rectangular* iteration spaces so that they achieve high performance on modern microprocessors.

The primary contributions of this thesis are the following:

- The proposal of a general compiler algorithm to perform tiling at the register level in arbitrary iteration space shapes.
- The proposal of a very simple heuristic to make the tile decisions for the level register level when dealing with non-rectangular iteration space.
- The proposal of a new compiler algorithm to compute *exact* loop bounds when multilevel tiling is performed. This algorithm improves upon conventional techniques on its cost, therefore making possible the inclusion of multilevel tiling in production compilers.
- A study of the effects of (1) tiling only for the register level, (2) tiling only for the cache level and (3) tiling for both the register and cache levels and a performance comparison between hand-optimized codes and automatic-optimized codes.

1.6 RELATED WORK

Several other researchers have also worked on code transformation to enhance the data locality and to extract ILP. This section summarizes their work and relates it to ours.

The first work on compiler transformation for maximizing cache locality was from Gallivan et al. [48][49][50]. They present a technique to describe the amount of data that must be in the cache for reuse to be possible. They call this the *reference window*. The window for a dependence describes all of the data that is brought into the cache for the two references from the time that one datum is accessed at the source until it is used again at the sink. A family of reference windows for an array represents all of its elements that must fit in the cache to capture all of reuse. To determine if the cache is large enough to hold every window, the window sizes are summed and compared against the size of the cache. If the cache is too small, blocking transformation can be used to decrease the size of the reference window. Their work basically is focused on analysis of cache and local memory behavior in loop nests to decide if loop tiling is profitable, and they do not give an algorithm for choosing the best tile sizes. Additionally, they do not explain how the tiled code is generated.

Wolfe's memory performance work [125][126][127][128] has concentrated on developing transformations to reshape loops to improve their cache performance. His loop transformation approach consists of individual loops and statements which can be transformed by a sequence of simple transformations that operate on one or two loops at time. The effectiveness of applying a specific transformation to a given goal is usually easy to measure. Unfortunately, it is often the case that several transformation will need to be applied successively. In this case, the best first transformation to apply is not obvious and the search of the best sequence of transformations must be performed exhaustively. The weakness in his approach is the lack of a decision algorithm to choose which transformation to apply to a nest. Wolfe also shows how tiling (or blocking) can be used to improve the memory performance of program loops and shows that his techniques for advanced loop permutation can be used to tile loops with non-rectangular iteration spaces and loops that are not perfectly nested. In particular, he discusses blocking at the cache level for triangular and trapezoidal shaped iteration spaces, but he does not present an algorithm and does not discuss tiling at the register level.

Wolf et al. [121][122][123] use an alternate approach to Wolfe's proposal based on matrix transformations. They model loop transformations such as loop permutation, reversal and skewing as unimodular transformations on the iteration space. A compound transformation is just a unimodular transformation, being a product of several elementary transformations. This model makes it possible to determine the best compound transformation directly by maximizing some objective function. They also provide a framework for determining memory usage within loop nests and use that framework to apply loop permutation, loop skewing, loop reversal and loop tiling. They give algorithms for computing the tile size and the best order of the tiles and present a method for determining the bounds of a transformed loop nest after applying a unimodular transformation or loop tiling. Their "compute bound" algorithm is very fast and simple. However the resulting code may contain redundant bounds and the loop bounds are not exact. As mentioned before, not computing exact bounds and the presence of redundant bounds can be negative if the register level is being exploited. Their method is able to perform tiling on several levels of the memory hierarchy, but, at the register level, they only exploit data reuse in one dimension of the iteration space and indicate that tiling more than one dimension is not trivial for complex loop nests. Moreover, they do not propose a method to determine tile sizes at the register level.

Carr et al. [23][24][25][27] discuss promotion of array references into registers (scalar replacement), and tiling for the register level via the unroll-and-jam transformation. As mentioned before, unroll-and-jam enhances data locality at the register level in the unrolled dimension, and it can be applied repeatedly to several loops in a nest to exploit data reuse at several levels. Indeed, applying unroll-and-jam to several loops in a nest is comparable to applying loop tiling at the register level. They use unroll-and-jam for exploiting reuse at the register level and improving ILP and discuss how to implement unroll-and-jam in the presence of triangular and trapezoidal loop nests. For these complex cases, they give the code transformation directly. To this end, they use pattern recognition techniques on the bounds of the loops and when the loop bounds do not match one of his patterns, no general algorithm to split the iteration space into simple ones that could be recognized through patterns is presented. Moreover, to decide the loops to be unrolled-and-jammed, they examine the dependence graph and search which loops carry most true and input dependences. They do not consider other parameters such as iteration space shape.

Porterfield's dissertation [102] contains a study of compiler techniques for improving data cache performance. His work is focused on how the application of various transformations - fission, fusion, permutation, unrolling, loop tiling - change the cache behavior, but he does not study the register level. He also does not deal with the code updating phase. He was also the first to explore software controlled prefetching for uniprocessors [22][102]. He proposed a compiler algorithm for inserting prefetches into dense-matrix codes. Then, Mowry [93][94][95] extended his work by automating the whole process and exploring prefetching for multiprocessors. Techniques for tolerating memory latency such as software prefetching can be combined with loop tiling to cope with whatever memory latency cannot be reduced through locality optimizations.

Two main differences can be pointed out between this thesis and previous work. First, we mostly focus on the "code updating" phase of the optimizing compilers, instead of the "decision algorithms" that determine the best tile sizes and the order of the tiles. Second, we also pay special attention to the register level instead of the cache level while previous work in the literature regarding loop tiling has mostly focused on enhancing data locality at the cache level and exploiting coarse-grain parallelism.

1.7 THESIS ORGANIZATION

This work is organized as follows:

- Chapter 2 gives some background on the loop tiling transformation and presents transformations and assumptions on which our compiler strategies are based.
- Chapter 3 describes our algorithm to perform tiling for the register level in complex loop nests and presents our locality analysis to make the tiling decisions at this level. We also present experiments to validate the effectiveness of our algorithm and compare against other transformations performed by current production compilers.
- Chapter 4 describes our efficient implementation of multilevel tiling to compute exact bounds, that we refer to as Simultaneous Multilevel Tiling. We compare our implementation against conventional techniques in terms of complexity and redundant bounds generated and present some experimental results.
- Chapter 5 presents a detailed evaluation of loop tiling. First, we discuss and evaluate the effect of tiling for one memory level (cache or registers) and tiling for multiple memory levels. In the second part of this chapter, we compare our automatic-optimized codes against the vendor hand-optimized codes and we show how compiler technology can make it possible for complex numerical codes to achieve high performance on modern microprocessors.
- Finally, Chapter 6 summarizes the contributions of this thesis and present open areas for future research.

2

LOOP TILING BASICS

Summary

In this chapter we gives some background on the loop tiling transformation. We review previous work related to loop tiling and presents the framework and assumptions on which our compiler strategies are based on. The chapter is organized as follows: In Section 2.1 we explain in what loop tiling consists and illustrate it with an example. In Section 2.2 we give some preliminary concepts needed for a better understanding of the following sections. Section 2.3 explains the three challenges that confront a compiler to apply loop tiling to a loop nest: dependence analysis, locality analysis and generating the transformed code. Finally, Section presents the assumptions on which our compiler strategies are based on and Section 2.5 summarizes this chapter.

2.1 THE LOOP TILING TRANSFORMATION

Loop Tiling is a well known loop transformation that a compiler can use to automatically create block algorithms. The advantage of block algorithms is that, while computing within a block, there is a high degree of data locality, allowing better register, cache or memory hierarchy performance [1][39][48][67][121][127]. Tiling is also a good paradigm for multiprocessors [123][128]. If tiling can be done so that different blocks are independent of each other, then different blocks can be assigned to different processors. In this chapter, however, we only focus on the loop tiling ability to enhance data locality.

To illustrate tiling and its importance, we present a simple example program and show how loop tiling enhances data locality. The example used is a matrix multiplication program, shown in Fig. 2.1a. In this code, the same row of A is used repeatedly by iterations of the middle (J) loop. If N is large relative to the cache size, so that an entire row of an array does not fit into the cache, then elements of the row may not be in the cache between reuses. For the D matrix, reuses of array elements occur in the outermost (I) loop. Between reuses of elements in D, the whole array is brought into the cache, which means that references to D will not hit in the cache. In the case of improving register utilization, it suffices to note that a register file is typically much smaller than N, so that the nest will not significantly benefit from register allocation of elements of either D or A.

Loop tiling consists of dividing the iteration space defined by the loop structures into regular tiles (or blocks) of some size and shape (typically squares or cubes), and then traversing the tiles to cover the whole iteration space. Loop tiling alters the order in which individual iterations are executed so that iterations from loops of the outer dimensions are executed before completing all the iterations of the inner loop. Thus, the distance between successive references to the same memory location is shortened and the probability of finding the associated data in the memory level being exploited is higher.

(a) Original code

```
do I = 1, N
  do J = 1, N
    do K = 1, N
      C(I,J) = C(I,J) + A(I,K) *D(K,J)
    enddo
  enddo
enddo
```

(b) Tiled code

```
do JJ = 1, N, B_JJ
  do KK = 1, N, B_KK
    do I = 1, N
      do J = JJ, min(N, JJ+B_JJ-1)
        do K = KK, min(N, KK+B_KK-1)
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo
      enddo
    enddo
  enddo
enddo
```

Figure 2.1: (a) Original code of matrix multiplication C=A x D. (b) Code after loop tiling.

Figure 2.1b shows the matrix multiplication program after loop tiling. Tiling reduces the number of intervening iterations between data reuses. Therefore, reuses of data occur more closely in time and the amount of data fetched between data reuses is reduced. This allows reused data to still be in the cache or register file, and hence reduces memory accesses. The tile size B_{JJ} x B_{KK} can be chosen to allow maximum reuse for a specific level of the memory hierarchy.

Figure 2.2a shows how data items are accessed in the non-tiled matrix multiplication previously shown in Fig. 2.1a. The same element C(I,J) is used by all iterations of the innermost (K) loop; it can be register allocated and is fetched from memory only once. The same row of A accessed in the innermost loop is reused in the next iteration of the middle (J) loop, and the same column of D is reused in the outermost (I) loop. Whether the data remains in the cache at the time of reuse depends on the size of the cache. Unless the cache is large enough to hold at least one NxN matrix, the data D will have been displaced before reuse. If the cache cannot hold even one row of the NxN matrix, then A data in the cache will also not be reused. In this latter case (worst case), $2N^3+N^2$ words of data need to be read from main memory in N^3 iterations. This high ratio of memory fetches to numerical operations significantly degrades machine performance, since memory fetches are of high latency.

By comparison, Fig. 2.2b shows how data items are accessed in the tiled matrix multiplication of Fig. 2.1b. Essentially, the patterns are the same as before, but operations are only performed on a B_{JJ} x B_{KK} submatrix of D. If B_{JJ} and B_{KK} are chosen properly, this submatrix fits in the cache and can be reused over and over. This allows all three matrices to have excellent reuse; ignoring interferences in the cache, the total main memory words accessed will be $(N^3/B_{JJ})+(N^3/B_{KK})+N^2$, which is an improvement of about a factor of B_{JJ} (assuming $B_{JJ} \approx B_{KK}$) over the non-tiled case.

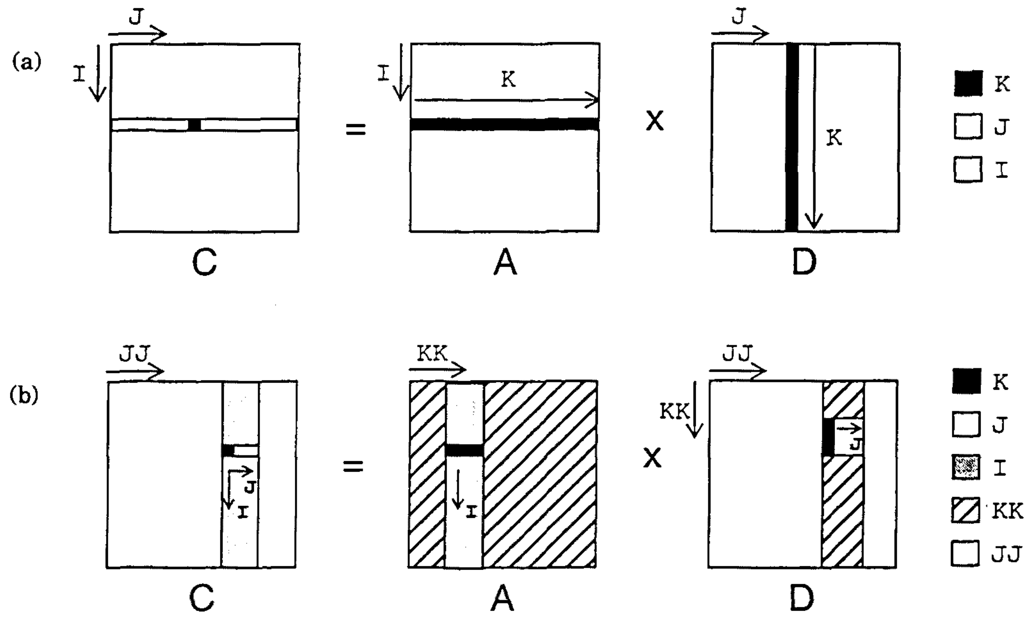


Figure 2.2: Data access pattern in (a) untiled and (b) tiled matrix multiplication.

Loop tiling is a powerful transformation that enhances locality of reference. Although we have only shown how loop tiling enhances locality at the cache level, loop tiling can also be applied at other memory levels, including the register file. For instance, by taking a small blocking size such that the block can be held in registers, we can reduce the number of loads and stores in a program. Chapter 3 deals with tiling for the register level.

2.2 PRELIMINARY CONCEPTS

In this section, we give a brief overview of some important concepts that are used throughout this thesis. In particular, concepts that are useful for program analysis and program restructuring.

Loop Nest Representation

We consider algorithms specified by a number of nested loops with the following form:

```
do $I_1 = L_1, U_1$
  do $I_2 = L_2, U_2$
    . . .
      do $I_n = L_n, U_n$
        loop body
enddo
```

The bounds of the loops are of the form:

$$L_i = max(l_{i,0}, l_{i,1}, l_{i,2}, \ldots) \qquad U_i = min(u_{i,0}, u_{i,1}, u_{i,2}, \ldots)$$

and $l_{i,j}$, $u_{i,j}$ are linear functions that can be expressed as follows:

$$l_{i,j} = \sum_{k=1}^{i-1} a_{i,j}^k \cdot I_k + \Upsilon_{i,j} \qquad u_{i,j} = \sum_{k=1}^{i-1} b_{i,j}^k \cdot I_k + \varphi_{i,j}$$

where $a_{i,j}^k, b_{i,j}^k \in Z$ $(1 \le k \le i-1)$, $\Upsilon_{i,j}$ and $\varphi_{i,j}$ are constants or parameters (problem size) and I_k $(1 \le k \le i-1)$ are loop control variables.

Iteration Space

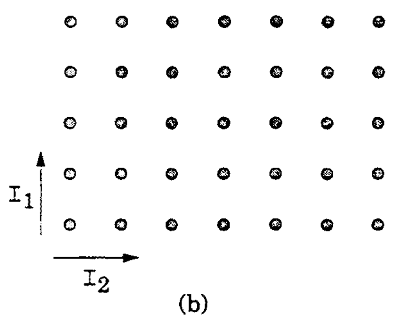
Any single iteration of the n-deep loop nest can be represented by a n-dimensional vector,

$$\vec{I} = (I_1, \ldots, I_n)^t$$

where each component of this vector is associated with one of the loops. The leftmost component of the vector is associated with the outermost loop and the rightmost component with the innermost loop.

```
do I1 = 0, 4
  do I2 = 0, 6
    A(I1,I2) = A(I1,I2) + A(I1-1,I2) + D(I1,I2+1)
    D(I1,I2) = D(I1-1,I2+1) + A(I1,I2)
  enddo
enddo
```

(a)



(b)

Figure 2.3: (a) Example loop nest. (b) Bounded Iteration Space (BIS).

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):

$$\text{BIS} = \{\vec{I} = (I_1, \dots, I_n)^t \,|\, (L_1 \le I_1 \le U_1, \dots, L_n \le I_n \le U_n)\}$$

Figure 2.3 shows an example of a loop nest (2.3a) and the corresponding BIS (2.3b).

The BIS can be specified in matrix form [54][77]. Taking into account the semantics of a do-loop, each of the linear functions $l_{i,j}$ and $u_{i,j}$ defines an inequality of the form:

$$\sum_{k=1}^{i-1} a_{i,j}^k \cdot I_k + \Upsilon_{i,j} \le I_i$$

$$I_i \le \sum_{k=1}^{i-1} b_{i,j}^k \cdot I_k + \varphi_{i,j}$$

Putting all these inequalities together, the following matrix inequality can be built:

$$A \cdot \vec{I} \le \beta$$

Every row of matrix A defines a lower bound component $l_{i,j}$ (or an upper bound component $u_{i,j}$) and it is built from coefficients $a_{i,j}^k$ and -1 (or $-b_{i,j}^k$ and 1). The n elements of vector $\vec{I}$ are the iteration control variables I_i, and β is a vector whose components are the coefficients $-\Upsilon_{i,j}$ (or $\varphi_{i,j}$).

As an example, the bounds of the BIS shown in Fig. 2.3 are represented by the following matrix inequality:

$$\begin{matrix} 0 \le I_1 & I_1 \le 4 \\ 0 \le I_2 & I_2 \le 6 \end{matrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \le \begin{bmatrix} 4 \\ 6 \\ 0 \\ 0 \end{bmatrix}$$

Data Dependences

Data dependence relations are used by the compiler to represent the essential ordering constraints among statements and the reuse of values in a program. We say that a dependence exits between two references if there exits a control-flow path from the first reference to the second and both references access the same memory location [76]. A dependence is

- a *true dependence* if the first reference writes to the location and the second reads from it,
- an *anti-dependence* if the first reference reads from the location and the second writes to it,
- an *output dependence* if both references write to the location, and
- an *input dependence* if both references read from the location.

When a code is transformed for optimization, the relative order of reads and writes to a particular memory location must be preserved; if not, the transformed code could produce incorrect results. In particular, true, anti and output dependences must be preserved. Input dependences, however, do not need to be preserved, but it is important to identify them when evaluating reuse of values in a program. Next, we introduce the terminology and notation used to represent data dependence relations in loops.

In loops, the data dependence relations are represented by n-dimensional vectors called dependence vectors: $\vec{d} = (\delta_1, \ldots, \delta_n)^t$.

Each component of a dependence vector is associated with one of the n nested loops, from the outermost to the innermost loop (left to right). The dependence vectors can provide different types of information (distances $(\delta_i \in Z)$, directions (δ_i ={<, >, =, ✱}), etc.) depending on the goals of the dependence analysis [125].

If the dependence vector provides distances, then each component of the vector is the number of iterations of the corresponding loop that separates two consecutive accesses to the data item associated to the dependence. For example, given the loop nest of Fig. 2.3a, the true dependence from $A(I_1,I_2)$ to $A(I_1\text{-}1,I_2)$ has a distance vector of (1,0), the true dependence from $D(I_1,I_2)$ to $D(I_1\text{-}1,I_2+1)$ has a distance vector of (1,-1), the anti-dependence from $D(I_1,I_2+1)$ to $D(I_1,I_2)$ has a distance vector of (0,1) the true dependence from $A(I_1,I_2)$ of the first statement to $A(I_1,I_2)$ of the second statement has a distance vector of (0,0) and the input dependence from $A(I_1,I_2)$ of the second statement to $A(I_1\text{-}1,I_2)$ of the first statement has a distance vector of (1,0).

If the dependence vector provides directions, then each component of the vector is the sign of the data dependence distance. Using the notation proposed by Wolfe [125], a forward direction "<" means that the dependence crosses an iteration boundary forward (from iteration i to iteration i+1, for example), that is, the sign of the distance is positive. A backward direction ">" means that the dependence crosses an iteration boundary backward (from iteration i to iteration i-1), that is, the sign of the distance is negative. An equal direction "=" means that the dependence does not cross an iteration boundary, that is, the dependence distance is zero. And an asterisk "*" is used when the direction is unknown or when all three of <, >, = apply. This annotation for data dependence relations is simpler but less precise than distance vectors.

The loop associated with the outermost (leftmost) non-zero distance vector entry is said to be the *carried loop* and if all distance vector entries are zero, the dependence is *loop independent*. In the example above, there are three dependences carried by loop I_1, one dependence carried by I_2 and one loop independent dependence.

Given a loop nest, all dependence vectors are always lexicographically positive $\left(\vec{d} \geq \vec{0}\right)$. This means that, if the dependence vector provides distances, then, the first non null component of the vector is positive and, if the dependence vector provides directions (following the previous notation), the first component different from =, must be equal to <.

Another concept is the *iteration space dependence graph* [129] that represents the constraints that prevent reordering of iterations of a loop. The iteration space associated with a loop contains one point for each iteration of the loop. If any statement in one iteration of the loop depends on any statement in a different iteration of the loop, that is, if there is a true, anti or output dependence that is not loop independent, the dependence would be represented by an arrow from the source iteration to the target iteration (input dependences do not prevent reordering of iterations). The iteration space dependence graph of the example in Fig. 2.3a is shown in Fig. 2.4. Note that the original execution order preserves dependences.

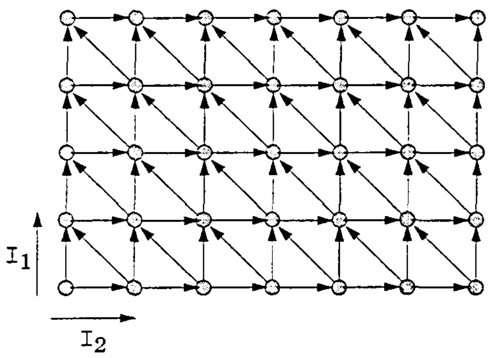
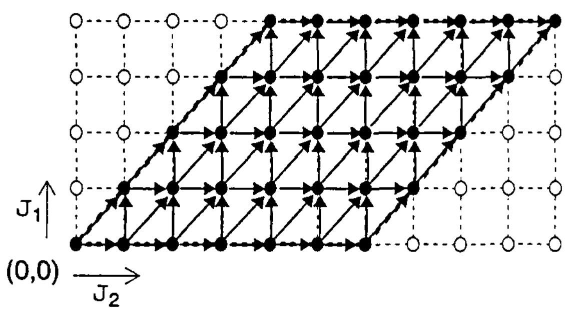


Figure 2.4: Iteration space dependence graph of the code in Fig. 2.3a



$T_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad |T_1| = 1$

Figure 2.5: TIS obtained through the unimodular transformation T_1.

Unimodular Transformations

A unimodular transformation [13][121] is a transformation that can be represented by a unimodular matrix T (its determinant is ± 1) and maps each iteration ($\vec{I} = (I_1, ..., I_n)^t$) of a n-dimensional iteration space to one iteration ($\vec{J} = (J_1, ..., J_n)^t$) of the transformed n-dimensional iteration space (TIS):

$$\mathrm{TIS} = \{\vec{J} = T \cdot \vec{I} \mid \vec{I} \in Z^n\} .$$

Since T is unimodular, both T and T^{-1} map points with integer coordinates (integer points for short) into integer points. As an example, Fig. 2.5 shows a piece of the TIS obtained when we apply the unimodular transformation matrix T_1 to the iteration space shown in Fig. 2.4. Note that the dependence vectors in the transformed iteration space have also changed according to the transformation matrix T_1.

A unimodular transformation T is a *legal* transformation only if true, anti and output dependences are preserved, which means that dependences in the transformed loop nest must be lexicographically positive. Let D be the set of true, anti and output dependence vectors of a loop nest. A unimodular transformation T is legal if and only if:

$$\vec{d} \in D \ : T \cdot \vec{d} \geq \vec{0}$$

In the example above, T_1 is a legal transformation because the three true dependences and the anti-dependence are preserved:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \geq \vec{0} \qquad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \geq \vec{0} \qquad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \geq \vec{0} \qquad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq \vec{0}$$

The image of the bounds of the BIS when applying a transformation T can be put in a matrix form, using the transformation matrix T and the matrix inequality which represents the bounds of the BIS:

$$\left.\begin{array}{l} A \cdot \vec{I} \leq \beta \\ T \cdot \vec{I} = \vec{J} \end{array}\right\} \Rightarrow \begin{array}{l} A \cdot T^{-1} \cdot \vec{J} \leq \beta \\ \hat{A} \cdot \vec{J} \leq \beta \end{array}$$

Since the BIS is convex, the above matrix inequality also delimits a convex space. This space is the minimum convex space which contains all the points of the TIS whose antiimage belongs to the BIS. We refer to this minimum convex space as the *Bounded Transformed Iteration Space* (BTIS).

The bounds of the BTIS can be extracted from the matrix inequality $\hat{A} \cdot \vec{J} \le \beta$. There are two different approaches to compute the bounds of BTIS. One approach, proposed by Kuhn [77] and also used by Ancourt and Irigoin [7], computes the *exact* loop bounds in the transformed code. Recall from Chapter 1 that a loop nest has *exact* bounds if it never executes an empty iteration. They use the Fourier-Motzkin elimination algorithm [7][54][77] (shown in Appendix A) to find solutions to the systems of linear inequalities $\hat{A} \cdot \vec{J} \le \beta$. In this scheme, the system of inequalities is solved by projecting it onto a reduced number of unknowns and then eliminating, one by one, each unknown. The unknowns are the n loop control variables of vector $\vec{J}$ and the order in which these unknowns are solved is important to obtain the exact bounds. They must be solved from the innermost to the outermost loop.

Continuing with the same example, the bounds of the BTIS are represented by the following matrix inequality:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} J_1 \\ J_2 \end{bmatrix} \le \begin{bmatrix} 4 \\ 6 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 1 \\ -1 & 0 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} J_1 \\ J_2 \end{bmatrix} \le \begin{bmatrix} 4 \\ 6 \\ 0 \\ 0 \end{bmatrix}$$

By applying the Fourier-Motzkin elimination algorithm to this system the following bounds of the BTIS are obtained:

$$0 \le J_1 \le 4$$

$$J_1 \le J_2 \le 6 + J_1$$

The transformed loop nest must scan all the integer points of the BTIS. Since T is a unimodular matrix, all the integer points of the BTIS have an integer antiimage in the BIS, and therefore, the bounds given by the Fourier-Motzkin algorithm can be directly used to build the loop nest required to scan the BTIS. In Fig. 2.5, the black dots represent the points of the BTIS.

Ancourt's approach is effective but, for certain classes of loops, it can be extremely time consuming, since this approach leads to a large number of constraints and Fourier-Motzkin operates in a pairwise fashion over these constraints. However, as mentioned in Chapter 1, computing *exact* bounds is critical when multilevel tiling, including the register level, is being applied.

An alternative approach to compute the bounds of BTIS was proposed by Wolf in [121]. By using projection information about the system before applying the unimodular transformation, he is able to more directly generate correct bounds. His approach is simpler and faster than Ancourt and Irigoin's, but the loop bounds are not exact, that is, bounds on the non-innermost loops may be wider than necessary or contain unnecessary minima and maxima computations.

At last, when a loop nest is transformed by a unimodular transformation, it is also necessary to rewrite the body of the loop nest. Suppose we have the loop nest

```
do I1 = . . .
   do I2 = . . .
      . . .
         do In = . . .
            S(I1, I2, . . ., In)
enddo
```

to which we apply the unimodular transformation T. The transformation of the body S only requires I_i $(1 \le i \le n)$ be replaced by the appropriate linear combination of J_i $(1 \le i \le n)$, where J_i's are the loop iteration variables for the transformed loop nest; that is:

$$\begin{bmatrix} I_1 \\ \dots \\ I_n \end{bmatrix} = T^{-1} \cdot \begin{bmatrix} J_1 \\ \dots \\ J_n \end{bmatrix}$$

Performing this substitution is all that is required to correctly transform the loop body.

Loop permutation, reversal and skewing are three well known transformations that can be represented as unimodular matrices [121]. Moreover, because of the properties of unimodular transformations, combinations of them are also unimodular transformations.

2.3 AUTOMATIC LOOP TILING

Three challenges confront a compiler that tries to apply loop tiling to a loop nest. First, the loop tiling transformation must not alter the computation performed by the loop nest. Determining whether the transformation is legal or not is the domain of dependence analysis. Second, it is difficult to choose for a given loop nest which are the tile sizes and shapes at each memory level and the arrangement of the tiles with respect to each other that deliver the best performance. Determining these tiling parameters is the domain of locality analysis. And third, once the tiling parameters have been decided, the loop nest has to be transformed appropriately. Figure 2.6 shows a diagram that specifies the sequence of compiler phases required to perform loop tiling. In this section, we explain in more detail each of these three phases.

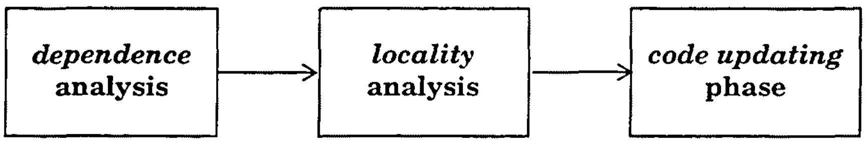


Figure 2.6: Compiler phases to perform loop tiling.

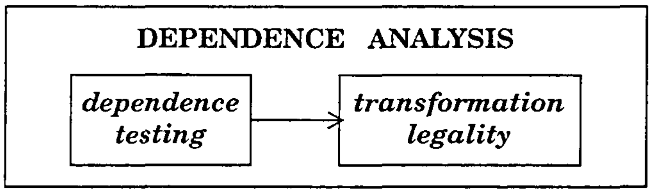


Figure 2.7: Phases of the dependence analyzer.

2.3.1 Dependence Analysis

The first challenge that confronts a compiler that tries to apply loop tiling to a loop nest is determining whether the transformation is legal or not. As mentioned before, a loop transformation is legal if true, anti and output dependences are preserved.

Two different phases can be distinguished in the dependence analysis. The first phase of the dependence analyzer consists in determining all array accesses that refer to the same array elements and computing their associated dependence distance (or direction) vectors. This task is known as *dependence testing*. Once all data dependence relations have been computed, the second step of the dependence analyzer is to determine whether a particular transformation is legal or not. Figure 2.7 shows a diagram of the dependence analyzer's phases.

Dependence Testing

An important point of the dependence analyzer in optimizing compilers is the accuracy, or the precision, of the dependence tests. In general, the dependence testing problem can be cast as an integer programming problem, where exact tests may be very expensive. There has been much work related to dependence testing that has considered different characteristics of the proposed tests such as cost (the speed of the test itself), applicability (subset of subscripts it is able to handle) and precision (number of false dependence computed) [14][51][77][86][89][103].

In optimizing compilers, it is very important to have dependence tests with high precision, since a false dependence can prevent the compiler from applying loop transformations that actually improve performance. For subscripted variables, the dependence analyzer must formulate a *data dependence system* and solve it as a system of linear equations and inequalities. The data dependence system is built with the dependence equations and a set of inequalities that add additional constraints to the system [129]. For instance, for a dependence to exist, a solution must lie within the limits of the induction variables imposed by the loop bounds. By considering the loop limit constraints, the dependence analyzer can reduce the number of false dependences and can compute the dependence distances and directions with more exactness, thus, increasing the opportunities to apply transformations such as loop tiling.

```
do J = 1, N
  do K = 1, N
    do I = 1, K-1
      D(I,J) = D(I,J) + D(K,J) * A(I,K)
enddo
```

Figure 2.8: Code of the STRMM problem, extracted from BLAS.

As an example, consider the code of the STRMM problem shown in Fig. 2.8. If the data dependence system does not include the loop limit constraints, the dependence analyzer must assume that there is a data dependence between D(I,J) and D(K,J) and that the corresponding direction vector is (=, <, ✱). The '✱' in the third dimension of this data dependence prevents the compiler from performing loop tiling. However, if the data dependence system includes loop limit constraints, it can be deduced that the direction of the dependence in dimension I is <, because one of the loop limit constraints is I ≤ K-1. Now, the direction vector of the dependence is (=, <, <) and this dependence does not prevent the compiler from performing loop tiling.

Transformation Legality

Loop tiling is a transformation that increases the depth of a loop nest. Given an n-deep nest, tiling can turn it into a nest that is anywhere from (n+1)-deep to 2n-deep, depending on how many of the loops are tiled. Tiling one or more loops of a loop nest rearranges its iteration space traversal so that it consists of a series of small polyhedra executed one after the other. Thus, loop tiling alters the order in which individual iterations are executed, and therefore, it cannot be a legal transformation for certain classes of loop nests. Figure 2.9 illustrates how loop tiling changes the order in which the original iterations are executed. The arrows in the figure represent the execution order.

Loop tiling is a legal transformation only if the n nested loops of the original code are *fully permutable* [121]. We say that n nested loops are *fully permutable* if all of the n! possible permutations of the n nested loops are legal. This means that, if dependence vectors provide distances, all components (δ_j with $1 \le j \le n$) of all dependence vectors (d_i) are greater or equal to 0 and, if dependence vectors provide directions, all components (δ_j) of all dependence vectors (d_i) are = or <.

We note that non fully permutable loop nests can always be turned into fully permutable nests by applying a unimodular transformation [121], such as loop skewing. This means that we can always find a unimodular transformation T so that $T \cdot \vec{d}$ is positive (i.e. all vector components are greater or equal to 0) for all dependence vectors.

Original loop nest

```
do I = 0, N
   do J = 0, N
      loop body
   enddo
enddo
```

Tiled loop nest

```
do II = 0, N, B_II
   do JJ = 0, N, B_JJ
      do I = II, min(II+B_II-1, N)
         do J = JJ, min(JJ+B_JJ-1, N)
            loop body
         enddo
      enddo
   enddo
enddo
```

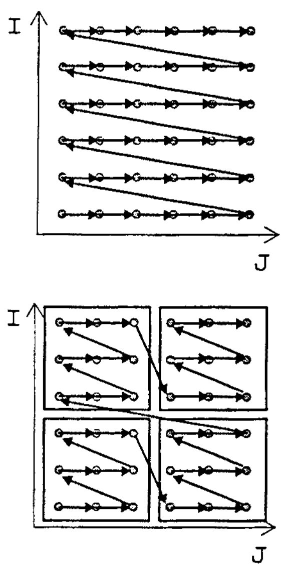


Figure 2.9: How loop tiling changes the order in which the iterations are executed.

Loop skewing is an unimodular transformation that changes the shape of the original iteration space. More precisely, skewing a loop J with respect to loop I by a factor f consists of changing the iteration vectors for each iteration (I, J) to (I, J+f*I). As an example, Fig. 2.10a shows again the non fully permutable loop nest used before in Fig. 2.3a and its iteration space dependence graph. Recall that the set D of dependence vectors (input dependences are not included) for this code is:

$$D = \{(1,0), (1,-1), (0,0), (0,1)\}$$

Note that the dependence vector (1,-1) implies that the loop nest is not a fully permutable nest. However, by skewing loop J by a factor f=1 with respect to loop I, the transformed loop nest becomes fully permutable. The set $\tilde{D}$ of dependence vectors in the transformed code is:

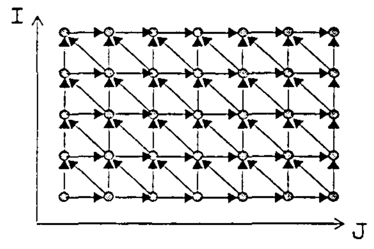
$$\tilde{D} = \{(1,1), (1,0), (0,0), (0,1)\}$$

Fig. 2.10b shows the loop nest after loop skewing and its iteration space dependence graph.

Note that loop skewing is always legal. It does not even change the execution order of the iterations, that is, iterations in the skewed iteration space are executed in the same order as the corresponding iterations in the original iteration space. It does only affect the data dependence vectors. Thus, although loop skewing does not generally improve code performance, it may enable other optimizations, such as loop tiling, that do so. However, for some kinds of loop nests, loop skewing can ruin data locality which means loop tiling won't be a profitable transformation. In these cases, it is preferable not to perform loop skewing and exploit data locality using a combination of other transformations that are less restrictive than loop tiling from the legality point of view.

(a) Original loop nest

```
do I = 0, 4
  do J = 0, 6
    A(I,J) = A(I,J) + A(I-1,J)+D(I,J+1)
    D(I,J) = D(I-1,J+1) + A(I,J)
  enddo
enddo
```



(b) Loop nest after loop skewing

```
do I = 0, 4
  do J = 0+I, 6+I
    A(I,J-I) = A(I,J-I) + A(I-1,J-I)+D(I,J-I+1)
    D(I,J-I) = D(I-1,J-I+1) + A(I,J-I)
  enddo
enddo
```

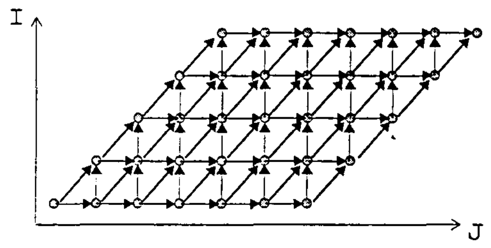


Figure 2.10: Loop nest and its iteration space dependence graph of (a) the original code, and (b) the transformed code.

2.3.2 Locality Analysis

Once the legality of the loop tiling transformation has been established, the second challenge that confronts a compiler is determining the tile sizes and shapes at each memory level and the arrangement of the tiles with respect to each other that deliver the best performance.

The first step to carry out this task is to identify data reuse within a loop nest. This field is the domain of reuse analysis. Several data reuse models can be found in the literature [23][71][83][121] that differ on the accuracy and applicability. This section shows how data reuse can be identified, using the reuse analysis proposed by Wolf [121] (we will use this reuse model later in Chapter 3). More precisely, we will explain the notion that reuse is carried by a specific loop. This reuse analysis is applicable to all memory hierarchy levels, although for concreteness we focus on the cache level. Once data reuse has been identified, the next step consists in determining the tiling parameters that deliver the best performance. In this section we also review previous work related to this second topic. Figure 2.11 shows the two main phases corresponding to the locality analysis.

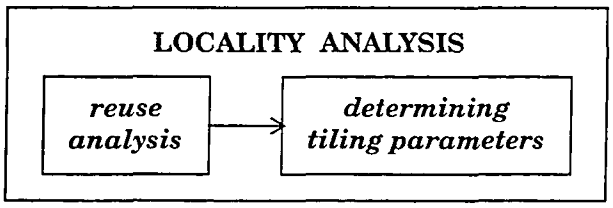


Figure 2.11: Phases of the locality analysis.

Reuse Analysis

We say that a data item is *reused* if the same data is used in multiple iterations of a loop nest. Thus, reuse is a measure that is inherent to the computation at hand and not dependent on the particular way the loop nest is written. Reuse may not lead to saving a main memory access if intervening iterations flush the data out of the cache between uses of the data. If the reused data actually remains in the cache, we say that the reference that enjoys the cache hit has *locality*. Therefore, it is important to realize that reuse does not necessarily result in locality. Instead, the references with data locality are a subset of the references with data reuse.

Reuse analysis attempts to discover those instances of array accesses that refer to the same cache line. There are four types of reuse: self-temporal reuse, self-spatial reuse, group temporal reuse and group spatial reuse. Self-temporal reuse occurs when a reference within a loop accesses the same data location in different iterations. Likewise, if a reference accesses data on the same cache line in different iterations, it is said to have self-spatial reuse. Furthermore, since a loop may have multiple references to the same array, different references may access the same locations. We say that there is group-temporal reuse if the references refer to the same memory location, and group-spatial reuse if they refer to the same cache line.

Wolf's reuse analysis represents the shape of the set of iterations that use the same data by a *reuse vector space*. He focuses on the common case of array references with affine subscript expressions, that is, subscript expressions that are affine functions of the loop induction variables. Each such array can be modeled by a q x n coefficient matrix H and an offset vector $\vec{c}$, where n is the depth of the loop nest and q is the dimensionality of the array. As an example, a reference A(I,J+1) in a 2-deep loop nest is represented by $A(H \cdot \vec{I} + \vec{c})$, where

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \vec{I} = \begin{bmatrix} I \\ J \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Let us first consider reuse that arises from single references. Given the previous representation, self-temporal reuse occurs between iterations $\overrightarrow{I_1}$ and $\overrightarrow{I_2}$ whenever $H \cdot \overrightarrow{I_1} + \vec{c} = H \cdot \overrightarrow{I_2} + \vec{c}$, i.e. when $H \cdot \left(\overrightarrow{I_1} - \overrightarrow{I_2}\right) = \vec{0}$. Rather than worrying about individual values of $\overrightarrow{I_1}$ and $\overrightarrow{I_2}$, we say that reuse occurs along the direction $\vec{r}$ when $H \cdot \vec{r} = \vec{0}$. The solution to this equation is the null-space of H, or *kernel* H, which is a vector space in R^n. If the null-space is empty, the reference does not exhibit self-temporal reuse.

Computing self-spatial reuse requires a slight variation of how self-temporal reuse is computed. Assuming data elements are stored in column major order, accesses to the same cache line will only occur when the same column is accessed. In addition, the column index expressions must be different, but still fall within the size of a cache line.

```
do I = 1, N
  do J = 1, N
    do K = 1, N
      C(I,J) = C(I,J) + A(I,K) * D(K,J)
    enddo
  enddo
enddo
```

Figure 2.12: Code of the matrix multiplication.

Thus, self-spatial reuse occurs between iterations $\vec{I_1}$ and $\vec{I_2}$ whenever all but the column index are equivalent. This is in contrast with self-temporal reuse, where all indices, including the column, must be equivalent. To extract this self-spatial reuse vector space, the row for stride-1 dimension in H (assuming column major order, it is the first row of H) is simply replaced with zeros to create H_s, and then we solve for the null-space of H_s. To check whether different elements are being accessed within the same column, it is necessary to compare whether the self-temporal and self-spatial reuse vector spaces are identical. This can occur since reusing the same data item is a degenerate case of reusing the same cache line. If the self-temporal and self-spatial reuse vector spaces are identical, then there is strictly self-temporal reuse. If they differ, then there is self-spatial reuse along the vectors that are unique to the self-spatial reuse vector space. Once accesses to different elements within the same column have been identified, the final step is to check whether the stride is less than the cache line size. If so, then the reference has self-spatial reuse.

As an example, consider the matrix multiplication loop nest shown in Fig. 2.12. The reference C(I,J) in the matrix multiplication nest yields

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{ and thus, } H_S = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} .$$

If we compute the null-space of H and H_S, we obtain: *kernel* H = span {(0, 0, 1)} and *kernel* H_S =span{(1,0,0), (0,0,1)}. Thus, we say that the reference C(I,J) has self-temporal reuse in direction K and self-spatial reuse in direction I. A similar analysis shows that reference A(I,K) has self-temporal reuse in direction J and self-spatial reuse in direction I, and that reference D(K,J) has self-temporal reuse in direction I and self-spatial reuse in direction K.

We now consider group reuse, reuse among different references. Wolf's model [121] focuses on *uniformly generated references* [50], that is, references whose array index expressions differ in at most the constant term. Two uniformly generated references can be written as $A(H \cdot \vec{I} + \vec{c_1})$ and $A(H \cdot \vec{I} + \vec{c_2})$.

Two distinct references $A(H \cdot \vec{I} + \vec{c_1})$ and $A(H \cdot \vec{I} + \vec{c_2})$ access the same data item if and only if

$$\exists \vec{r} : H \cdot \vec{r} = \vec{c_1} - \vec{c_2}$$

To determine whether such an $\vec{r}$ exists, the system of equations $H \cdot \vec{r} = \vec{c_1} - \vec{c_2}$ is solved to get a particular integer solution $\vec{r_p}$, if one exists. Then, the group-temporal reuse vector space is the null-space of H plus span $\{\vec{r_p}\}$.

Finally, to detect group-spatial reuse between two uniformly generated references, the previous equation must be slightly modified by replacing the first rows in H, $\vec{c_1}$ and $\vec{c_2}$, with zeros to form H_S, $\overrightarrow{c_{S,1}}$ and $\overrightarrow{c_{S,2}}$, respectively (remember that we assume data are stored in column major order). Thus, two distinct references A($H \cdot \vec{\mathrm{I}} + \vec{c_1}$) and A($H \cdot \vec{\mathrm{I}} + \vec{c_2}$) have group-spatial reuse if and only if

$$\exists \vec{r} : H_S \cdot \vec{r} = \overrightarrow{c_{S,1}} - \overrightarrow{c_{S,2}},$$

and also provided that the constant difference between the first rows of $\vec{c_1}$ and $\vec{c_2}$ is less than the cache line size divided by the element size. Then, the group-spatial reuse vector space is the null-space of H_S plus span $\{\vec{r_p}\}$, where $\vec{r_p}$ is a particular solution of the above system of equations.

As an example, consider the code of Fig. 2.13. The references A(2*I,J) and A(2*I,J+1) in the nest yield

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \qquad \vec{c_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \vec{c_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

In this example, *kernel* $H = \emptyset$ and $\vec{r_p} = (0, -1)$. Thus, the group-temporal reuse vector space is span{(0,-1)} and we say that there is group-temporal reuse between references A(2*I,J) and A(2*I,J+1) in direction J.

However, the references A(2*I,J) and A(2*I+1,J) in the nest yield

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \qquad \vec{c_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \vec{c_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

In this case, there is no integer solution $\vec{r_p}$, such that $H \cdot \vec{r_p} = \vec{c_1} - \vec{c_2}$, since there is no integer r_1 such that $2 \cdot r_1 = 1$. Thus, there is no group-temporal reuse between references A(2*I,J) and A(2*I+1,J). Nevertheless, there is an integer solution to the system $H_S \cdot \vec{r_p} = \overrightarrow{c_{S,1}} - \overrightarrow{c_{S,2}}$, where

$$H_S = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \qquad \overrightarrow{c_{S,1}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \overrightarrow{c_{S,2}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

that is $\vec{r_p} = (1, 0)$. Thus, given a cache line size of at least two array elements, group-spatial reuse does occur between references A(2*I,J) and A(2*I+1,J).

```
do I = 1, N
  do J = 1, N
    A(2*I,J) = A(2*I,J) + A(2*I,J+1) + A(2*I+1,J)
  enddo
enddo
```

Figure 2.13: Example loop nest.

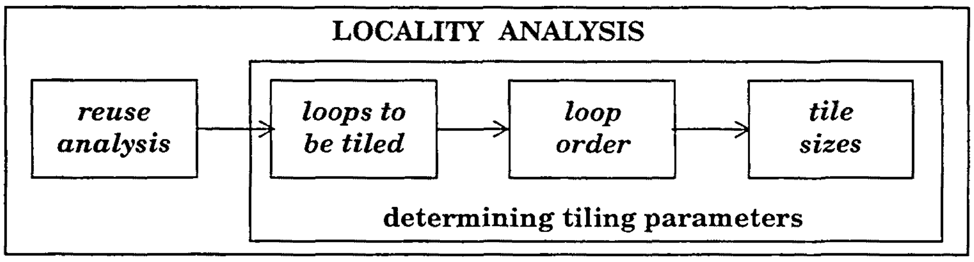


Figure 2.14: Expanded phases of the locality analysis.

Thus far, we have concentrated on reuse analysis of loop nests in the presence of caches. Considerations for registers, local memory, virtual memory and so on are similar, and the differences between these memory hierarchy levels are simple enough to be easily taken into account. For example, registers typically cannot take advantage of spatial reuse, so we can only identify temporal reuse at this level. On the other hand, exploiting spatial locality is very important when optimizing for virtual memory, where the "line size" is the page size, which may be very large.

We want also to outline that for loop nests having complicated data access pattern, data reuse can occur along directions not parallel to the iteration space axes. For example, in banded matrix computations [4][33][40][52], arrays are usually traversed along the diagonals, thus consecutive referenced elements are not in the same cache line. This yields large working sets and little reuse of cache lines, clearly a bad situation if one desires to exploit cache locality. W. Li in [83] proposed a loop transformation called *height reduction* that improves cache locality by modifying the data access pattern such that reuse occurs along directions parallel to the iteration space axes. Thus, by applying height reduction, data accesses are within a small working set and cache lines are reused. For banded matrix problems, this transformation should be applied before loop tiling.

Determining Tiling Parameters

Now that we have shown how the various types of reuse can be computed, the next step in the locality analysis is determining the tiling parameters that maximize data locality. The tiling parameters are the loops to be tiled at each level, their tile sizes and the relative order of the loops. Figure 2.14 shows a diagram of the different phases of the locality analysis.

Previous work on determining the tiling parameters has mostly targeted the cache level, while less attention has been paid to the register level. Although the basic principles in memory hierarchy optimization are similar for all levels, each level has slightly different characteristics, requiring slightly different considerations. For instance, caches usually have low set associativity, so cache conflicts can cause desired data to be replaced. The choice of tile size for cache optimization is actually more difficult than for other memory hierarchy levels, because the data mapping and the cache replacement policy makes it difficult to predict whether a data item will be in the cache. Another major difference between cache and registers is their capacity. For registers, the tile size is smaller than for caches, since registers set size is smaller than the cache size.

One step of the locality analysis is determining which loops of a nest should be tiled for maximizing data locality. Exploiting data reuse in more than one dimension of the iteration space, whenever possible, improves the performance of the memory hierarchy [98][122]. To determine which loops should be tiled, it is necessary to quantify the data locality provided by various permutation of the loop nest to select the permutation that yields the maximum locality. In [122], the authors provide a framework to quantify data locality at the cache level and propose an algorithm to select the loops to be tiled.

Once the loops to be tiled have been decided, the relative order of the loops must be determined. If we are tiling only for cache locality, for example, the loops that traverse the iterations inside a tile should be reordered to enhance temporal locality as much as possible, which may allow register allocation to resolve some of the data references from values already in registers. Spatial locality in these loops is also important for taking immediate advantage of a cache line. The order of the loops that traverse the tiles should be chosen to enhance reuse between tiles [83]. If we select the best arrangement of the tiles with respect to each other in the looping structure, we may be able to reduce the number of compulsory[1] misses for a tile by taking advantage of reuse between consecutive tiles.

Finally, given the loops to be tiled and the relative order of the loops in the tiled nest, the tile sizes have to be computed. We apply tiling to a loop nest to force it to operate on subarrays of any desired size. If the tile sizes are chosen optimally, each fragment of the work may be completed without any cache misses other than compulsory ones.

The organization of the data cache can make a dramatic difference in the overall effectiveness of tiling and on the effectiveness of a particular tile size for a given problem size. Suppose we have a direct-mapped cache and some particular loop nest. While the cache may be large enough to hold most of the data being processed, it may not be able to do so because of interferences between contents of different arrays (cross-interference), or even between two parts of the same array (self-interferences). Even if we fully tile the loop nest, sometimes we need to make the tile sizes surprisingly small compared to the size of the cache to avoid interferences. Having a set-associative cache obviously reduces the frequency of such interferences, but the more arrays we are processing in a single loop nest, the less effective it does so. Choosing a fixed tile size independent of the array sizes can prove disastrous, especially for direct mapped caches [80], because a fixed tiles size can dramatically increase interferences. Meanwhile, if we allow variable tile sizes, it is possible to choose an appropriate combination of array sizes and tile sizes that can be very effective for reducing conflict misses, while changing the array sizes by as little as 1% or 2%. Thus, it is essential that tile sizes be allowed to change with the array sizes.

1.Compulsory misses occur when a cache line is referenced for the first time

Several works have been done in this domain. Lam et. al. [80] present an algorithm for computing the tile size, which selects the largest square tile that does not incur self-interference conflicts. Esseghir [42] presents a tile size selection algorithm for one-dimensional tiling, which chooses the maximum number of complete columns that fit in the cache. This algorithm leaves one large gap of unused cache, it cannot exploit the benefits of multi-dimensional tiling, and also does not consider cross-interference among arrays. Coleman et. al. [31] present a technique based on the Euclidean g.c.d. computation algorithm for selecting a tile size that attempts to maximize the cache utilization while eliminating self-interferences within the tile. They also try to minimize cross-interferences conflicts. Finally, Moon and Saavedra in [92] introduce hyperblocking or hypertiling, which is a novel optimization technique that reorganizes data arrays with the goal of eliminating self and cross interference misses of tiled loop nests. In their experiments, they compared hyperblocking against other proposed heuristics for tile size selection [31][42][80] and show that their technique can effectively eliminate cache conflicts of tiled loop nests while incurring insignificant overheads in reorganizing the arrays.

Our discussion so far has mostly been in terms of tiling for cache memory locality. In fact, the memory hierarchy is quite a bit deeper. The processor usually has a register file, multiple levels of cache, as well as virtual memory level. If the loops can be tiled for locality at the cache level, they can also be tiled for locality at the register level, secondary cache level, virtual memory level, etc. In particular, if the primary cache is very small, tiling for primary and secondary cache locality as well as for the register level, may yield the best performance. Moreover, tiling or otherwise optimizing for virtual memory performance might also be necessary for very large data sets [1].

To enhance data locality at several levels of the memory hierarchy, *multilevel tiling* has to be performed. Multilevel tiling consists in recursively applying tiling for each level by dividing a tile of a higher level into subtiles [28][97][121]. Each level of tiling exploits one level of the memory hierarchy.

When multilevel tiling is performed, the interaction between different levels must be considered. Achieving the optimization of only one level of the hierarchy is simple, whereas the overall optimization for several levels is complex [90]. In [97][98] Multilevel Orthogonal Block (MOB) forms are proposed to achieve a high degree of data reuse in all levels of the memory hierarchy. The basic rule in the construction of a MOB form is that the direction of tiles in adjacent levels should be different. The direction of a tile is determined by the loop that is not tiled for this level. Furthermore, the orthogonality property of the MOB forms allows a "sequential" optimization to determine the order in which tiles are traversed and the size of the tiles level by level. Thus, optimizing for multiple memory hierarchy levels is simply a matter of optimizing one level at a time.

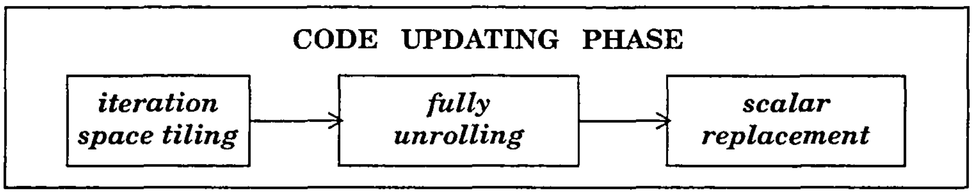


Figure 2.15: Steps of the compiler's updating phase.

2.3.3 Code Updating Phase

Finally, once the tile sizes and shapes at each memory level and the arrangement of the tiles have been determined, the loop nest has to be transformed appropriately. In this section we will describe the transformation steps carried out by the *code updating phase* of the compiler to generate the multilevel tiled loop nest.

We will distinguish between the implementation of tiling for the register level and the implementation of tiling for other memory levels. In both cases, the iteration space is divided into regular tiles of the same size an shape, however, the implementation of tiling for the register level requires an extra phase not needed for other memory levels. Since registers are only addressable using absolute addresses (the register number), after dividing the iteration space into tiles, it is necessary to fully unroll the loops that traverse the iterations inside the register tiles. Thus, reused array elements are exposed in the new unrolled loop body.

At last, scalar replacement can be used to expose reuse between iterations of the innermost loop and, therefore, to reduce loads and stores in the loop body. This transformation should always be done, independently whether tiling is being applied for the register level or not.

Thus, we can differentiate three different steps in the code updating phase when multilevel tiling is being applied. First, the iteration space is divided into different levels of tiles (one level of tiling exploits one level of the memory hierarchy). We will refer to this step as the iteration space tiling phase. The second step of the updating phase consists of fully unrolling the loops that traverse the iterations inside the register tiles. Obviously, this second step is only performed if multilevel tiling includes the register level. Finally, the third step consists in applying scalar replacement to expose reuse between iterations of the innermost loop. Figure 2.15 shows a diagram of the three steps performed by the compiler's updating phase.

Iteration Space Tiling

Let's first show how to carry out the iteration space tiling phase when tiling is being applied only for one level of the memory hierarchy. Conventional tiling techniques implement one level of tiling combining two well-known transformations: strip-mining and loop permutation [128].

Strip-mining is used to partition one dimension of the iteration space into strips. It decomposes a single loop into two nested loops; the outer loop (the *tile* loop) steps between strips of consecutive iterations, and the inner loop (the *element* loop) traverses the iterations within a strip. The loop bounds after strip-mining a loop are directly obtained by applying the formula of strip-mining. The general formula for strip mining a loop is given by the expression of Fig. 2.16,

$$\text{do I = L, U} \xrightarrow{\textbf{strip-mining}} \begin{array}{l} \text{do II} = \lfloor (\text{L} - \text{oft}_{\text{II}}) / \text{B}_{\text{II}} \rfloor * \text{B}_{\text{II}} + \text{oft}_{\text{II}}, \text{U}, \text{B}_{\text{II}} \\ \quad \text{do I} = \max(\text{II}, \text{L}), \min(\text{II}+\text{B}_{\text{II}}-1, \text{U}) \end{array}$$

Figure 2.16: Formula for strip-mining a loop.

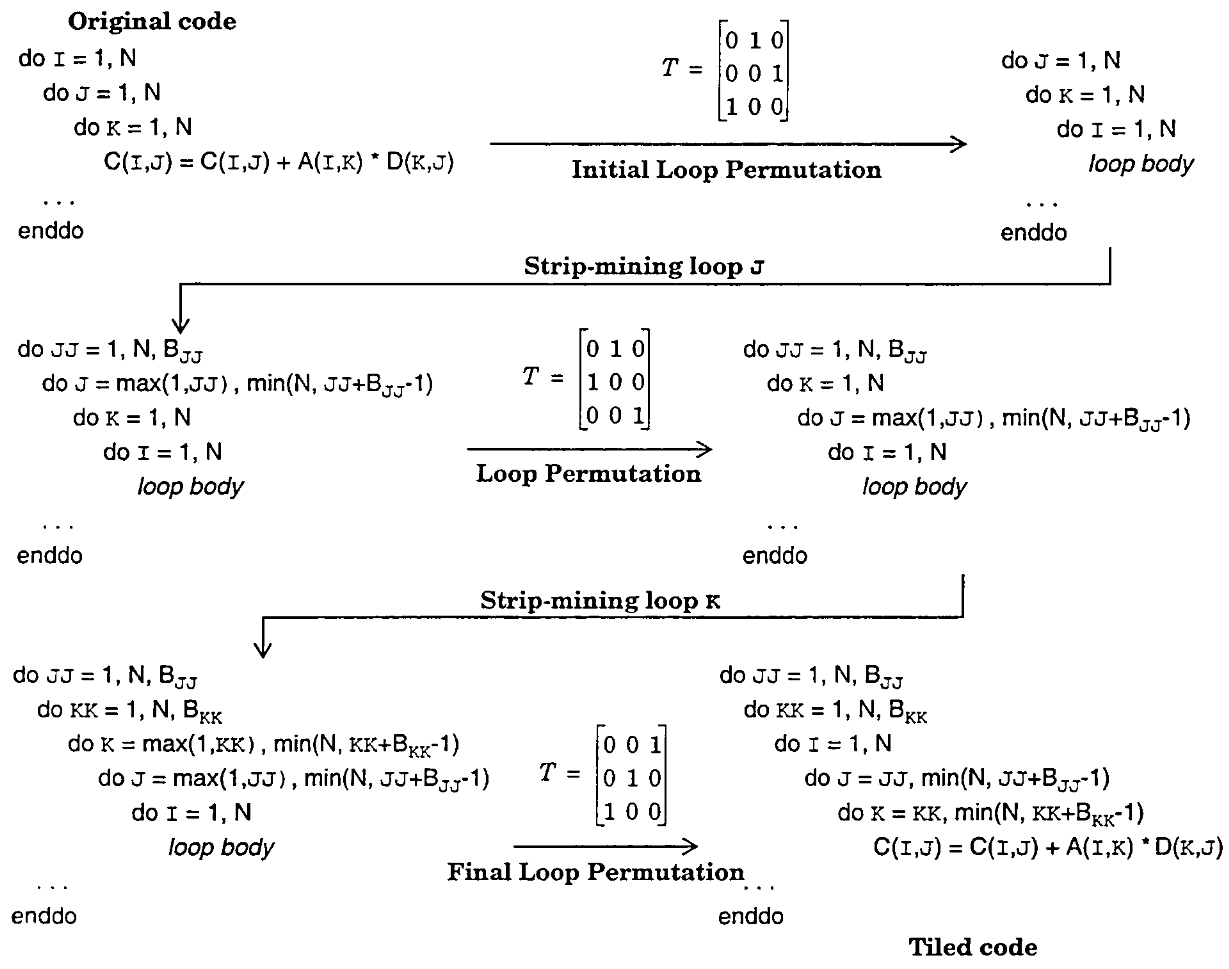
where II is the tile loop, I is the element loop, B_{II} is the strip size and $oft_{II} \in Z$ $(0 \le oft_{II} < B_{II})$ is an offset. Using this expression, the strip boundaries are always parallel to the iteration space axes and the offset determines the origin of the first strip [129].

Loop permutation is used to establish the order in which the iterations inside the tiles are traversed. Loop permutation is a unimodular transformation [121] that generalizes the loop interchange transformation. Loop interchange is the transformation that reverses the order of two adjacent loops in a nest. Loop permutation, instead, allows more than two loops to be moved at once and does not require them to be adjacent. A permutation σ on a loop nest transforms iterations $(I_1, ..., I_n)$ to $(I_{\sigma 1}, ..., I_{\sigma n})$. This transformation can be expressed in matrix form as I_{σ}, the n x n identity matrix I with the rows permuted by σ. For example, to permute the loop nest IJK of Fig. 2.1a (page 36) into JKI, the following unimodular transformation T has to be applied:

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

The loop bounds after a loop permutation can be obtained using the theory of unimodular transformations [122] and, to compute the exact bounds, the Fourier-Motzkin Elimination algorithm is used when the permutation unimodular matrix is applied [16][7][77] (see Section 2.2).

To exploit data reuse in several dimensions of the iteration space, multi-dimensional tiling has to be performed, that means that the iteration space has to be partitioned in more than one dimension. In this case, conventional techniques apply strip-mining and loop permutation repeatedly, as many times as dimensions have to be partitioned. An initial loop permutation must be performed if the loop order of the original loop nest is not such that the outermost loop is the loop to be strip-mined first. For each space dimension to be partitioned, conventional techniques apply first strip-mining to the outermost element loop, and then, a loop permutation is performed to order the inner element loops such that the next loop to be strip-mined becomes the outermost of them. Notice that the loops involved in a loop permutation are always the innermost loops that have steps equal to one and therefore, they define a convex iteration space. Thus, the theory of unimodular transformations can be



Original code

```
do I = 1, N
  do J = 1, N
    do K = 1, N
      C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \xrightarrow[\textbf{Initial Loop Permutation}]{}$$

```
do J = 1, N
  do K = 1, N
    do I = 1, N
      loop body
...
enddo
```

Strip-mining loop J

```
do JJ = 1, N, B_JJ
  do J = max(1,JJ), min(N, JJ+B_JJ-1)
    do K = 1, N
      do I = 1, N
        loop body
...
enddo
```

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow[\textbf{Loop Permutation}]{}$$

```
do JJ = 1, N, B_JJ
  do K = 1, N
    do J = max(1,JJ), min(N, JJ+B_JJ-1)
      do I = 1, N
        loop body
...
enddo
```

Strip-mining loop K

```
do JJ = 1, N, B_JJ
  do KK = 1, N, B_KK
    do K = max(1,KK), min(N, KK+B_KK-1)
      do J = max(1,JJ), min(N, JJ+B_JJ-1)
        do I = 1, N
          loop body
...
enddo
```

$$T = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \xrightarrow[\textbf{Final Loop Permutation}]{}$$

```
do JJ = 1, N, B_JJ
  do KK = 1, N, B_KK
    do I = 1, N
      do J = JJ, min(N, JJ+B_JJ-1)
        do K = KK, min(N, KK+B_KK-1)
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

Tiled code

Figure 2.17: Conventional implementation of one level of tiling. We show the order in which strip-mining and loop permutation are applied to perform one level of tiling for exploiting data reuse at two dimensions of the 3-dimensional iteration space (two-dimensional tiling). B_{JJ} and B_{KK} are the tile sizes in each dimension.

used to perform the loop permutation. After strip-mining all desired loops, a final loop permutation is required to order the inner loops as wished for the final code. In the final tiled code, the loops that step between tiles (*tile* loops) are the outer loops and the loops that step points within a tile (*element* loops) are the inner loops.

Figure 2.17 shows how conventional techniques implement one level of tiling. In the example, one level of tiling is applied to the matrix multiplication loop nest (shown in Fig. 2.1a) to obtain the tiled loop nest of Fig. 2.1b. The offset used in the strip-mining transformation is 1 (note that $\lfloor(1-1)/B\rfloor * B+1=1$) and, for every loop permutation performed, we specify its corresponding unimodular matrix. A two-dimensional tiling is performed to exploit data reuse at two dimensions of the iteration space. B_{JJ} and B_{KK} are the tile sizes in each dimension. In the final tiled code, the outer loops, which have steps different from one, are the loops that step between tiles and the inner loops, which have steps equal to one, are the loops that traverse the points within the tiles.

Finally, we note that it is not necessary to rewrite the loop body when performing loop tiling because (1) the strip-mining transformation does not modify the loop body and (2) although the loop permutation does, we use in the transformed code the same names for the loop iteration variables as in the original code, thus avoiding to rewrite the loop body.

Multilevel tiling has been implemented by applying tiling level by level [28][122], going from the outermost (i.e. virtual memory level) to the innermost level (i.e. register level). In Fig. 2.17 another level of tiling can be performed by applying tiling again to loops I, J and K of the resulting code.

Fully Unrolling

As mentioned before, the implementation of tiling for the register level requires an extra phase not needed for other memory levels. Since registers are only addressable using absolute addresses (the register number), after dividing the iteration space into register tiles, it is necessary to fully unroll the loops that traverse the iterations inside the register tiles, so that reused array elements are exposed.

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.

As an example, suppose that we want to perform two levels of tiling (i.e. cache and registers) to the matrix multiplication code (see Fig. 2.17). Suppose also that the locality analysis has decided that, at each level, data reuse has to be exploited in two dimensions of the iteration space. At the cache level dimensions J and K have to be tiled with tile sizes B_{JJJ} and B_{KKK}, respectively, and, at the register level, dimensions I and J are tiled with tile sizes B_{II} and B_{JJ}, respectively. After tiling the iteration space for two level as explained previously, we obtain the code of Fig. 2.18a. The length of the loop index variables refers to the level it is exploiting (JJ for register and JJJ for cache). For simplicity, we assume that N is multiple of B_{JJJ}, B_{KKK}, B_{II} and B_{JJ} and, in turn, B_{JJJ} is multiple of B_{JJ}.

To properly exploit the register level, the loops that traverse the iterations inside the register tile (loops I and J) should be fully unrolled. Since both loops execute exactly B_{II} and B_{JJ} iterations, respectively, they can be easy fully unrolled. Figure 2.18b shows the code after fully unrolling loops I and J, assuming B_{II}=B_{JJ}=2.

Scalar Replacement

At last, scalar replacement can be used to eliminate redundant loads and stores in the new unrolled loop body. Although conventional compilation systems do a good job of allocating variables to registers, their handling of subscripted variables leaves much to be desired. Most compilers fail to recognize even simplest opportunities for reuse of subscripted variables between iterations of the innermost loop. For example, in the code of Fig. 2.18b, most compilers will not keep C(II,JJ), C(II,JJ+1),

(a) iteration space tiling

```
do JJJ = 1, N, B_JJJ
  do KKK = 1, N, B_KKK
    do II = 1, N, B_II
      do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
        do K = KKK, KKK+B_KKK-1
          do I = II, II+B_II-1
            do J = JJ, JJ+B_JJ-1
              C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

(b) fully unrolling

```
do JJJ = 1, N, B_JJJ
  do KKK = 1, N, B_KKK
    do II = 1, N, B_II
      do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
        do K = KKK, KKK+B_KKK-1
          C(II,JJ) = C(II,JJ) + A(II,K) * D(K,JJ)
          C(II,JJ+1) = C(II,JJ+1) + A(II,K) * D(K,JJ+1)
          C(II+1,JJ) = C(II+1,JJ) + A(II+1,K) * D(K,JJ)
          C(II+1,JJ+1) = C(II+1,JJ+1) + A(II+1,K) * D(K,JJ+1)
...
enddo
```

(c) scalar replacement

```
do JJJ = 1, N, B_JJJ
  do KKK = 1, N, B_KKK
    do II = 1, N, B_II
      do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
        RR1 = C(II,JJ)
        RR2 = C(II,JJ+1)
        RR3 = C(II+1,JJ)
        RR4 = C(II+1,JJ+1)
        do K = KKK, KKK+B_KKK-1
          RR1 = RR1 + A(II,K) * D(K,JJ)
          RR2 = RR2 + A(II,K) * D(K,JJ+1)
          RR3 = RR3 + A(II+1,K) * D(K,JJ)
          RR4 = RR4 + A(II+1,K) * D(K,JJ+1)
        enddo
        C(II,JJ) = RR1
        C(II,JJ+1) = RR2
        C(II+1,JJ) = RR3
        C(II+1,JJ+1) = RR4
...
enddo
```

Figure 2.18: Loop nest (a) after tiling the iteration space for two levels, cache and register, (b) after fully unrolling the loops that traverse the iterations inside the register tile and (c) after applying scalar replacement.

C(II+1,JJ) and C(II+1,JJ+1) in registers in the inner K loop. This happens in spite of the fact that standard optimization techniques are able to determine that the addresses of the subscripted variables are invariant in the inner loop. The principal reason for the problem is that the data-flow analysis used by standard compilers is not powerful enough to recognize most opportunities for reuse of array variables. Array variables are treated in a particularly naive fashion making it impossible to determine when a specific element might be reused. Scalar replacement, proposed by [20][23], is a transformation that uses dependence information to find reuse of array values and expose it by replacing the references with scalar temporal variables. Figure 2.18c shows the code of Fig. 2.18b after applying scalar replacement. Note that scalar replacement is used to expose reuse of array values between different iterations of the innermost loop. For references such as A(II,K), A(II+1,K), D(K,JJ) and D(K,JJ+1), that exhibit reuse inside the loop body (but not between iterations) current compilers are already able to eliminate redundant loads or stores.

Note that loop tiling for the register level enhances data reuse inside the loop body. In the example above, it exploits the reuse of A(II,K), A(II+1,K), D(K,JJ) and D(K,JJ+1) inside the inner K loop. The data reuse exploited across iterations of the inner loop (matrix C) is achieved by the scalar replacement transformation and we note that both codes, the one not tiled for the register level (Fig. 2.17) and the one tiled for the register level (Fig. 2.18c), achieve the same degree of reuse of matrix C. In the cache tiled code of Fig. 2.17, even after scalar replacement, there are two loads and zero stores per iteration, whereas in the register tiled loop nest (Fig. 2.18c) there are 4 loads and zero stores per four iterations, which means only 1 load per iteration.

Transformation of Non-perfectly Nested Loops

Thus far, we have assumed that we are transforming a perfectly nested loop nest. That is, all computation is nested within the innermost loop. Now, we focus on loop nests that are not perfectly nested. It is possible to apply multilevel tiling to these loop nests as well. For instance, many important non-perfectly nested loop nests such as LU decomposition without pivoting, Cholesky factorization and Given QR factorization, can benefit from multilevel tiling.

When a loop nest is not perfectly nested, we can use a code sinking transformation [129] to convert it into a perfectly nested loop nest. 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. As an example, Fig. 2.19 shows how a general 3-deep non-perfectly nested loop nest is converted into perfectly nested.

The code sinking transformation is legal if and only if the loops in the nest never execute empty iterations, that means, that the bounds of the loops must be exact. This ensures that if a non-perfectly nested statement S or S' would have executed in the original (non-perfectly nested) code, then it still will in the transformed (perfectly nested) code. On the other hand, once the nest is transformed in this way, the data dependences can be extracted in a straightforward manner. By augmenting the set of

```
do $I_1 = L_1, U_1$
  $S_1$
  do $I_1 = L_2, U_2$
    $S_2$
    do $I_3 = L_3, U_3$
      $S_3$
    enddo
    $S'_2$
  enddo
  $S'_1$
enddo
```

$\xrightarrow{\textbf{code sinking}}$

```
do $I_1 = L_1, U_1$
  do $I_2 = L_2, U_2$
    do $I_3 = L_3, U_3$
      if ($I_3$.eq.$L_3$) .and. ($I_2$.eq.$L_2$) then $S_1$
      if ($I_3$.eq.$L_3$) then $S_2$
      $S_3$
      if ($I_3$.eq.$U_3$) then $S'_2$
      if ($I_3$.eq.$U_3$) .and. ($I_2$.eq.$U_2$) then $S'_1$
    enddo
  enddo
enddo
```

Figure 2.19: How to convert a 3-deep non-perfectly nested loop into perfectly nested.

constraints that the data dependence analyzer is solving [89] with the conditional constraints, the data dependence analyzer has the necessary information to determine all data dependences in the new perfectly nested loop nest.

Now that we have a set of perfectly nested loops, multilevel tiling can be applied as explained before. However, having conditional statements in the loop body can increase execution time because, in every iteration, the conditions of the if-statements must be evaluated. To avoid this time execution overhead, after tiling the iteration space (i.e. after the iteration space tiling phase and before unrolling the inner loops), the if-statements can be eliminated using index set splitting [17][73]. Index set splitting is used to partition the iteration space of the tiled loop nest, so that the tiled loop nest containing if-statements is restructured into a sequence of nested loops with no if-statements.

As an example, consider the set of non-perfectly nested loops of Fig. 2.20a that corresponds to the matrix multiplication code with initialization of matrix C. After converting the nest into perfectly

(a) Original code

```
do I = 1, N
  do J = 1, N
     C(I,J) = 0
     do K = 1, N
        C(I,J) = C(I,J) + A(I,K) * D(K,J)
 . . .
enddo
```

(b) Code sinking

```
do I = 1, N
  do J = 1, N
     do K = 1, N
        if (K.eq.1) C(I,J) = 0
        C(I,J) = C(I,J) + A(I,K) * D(K,J)
 . . .
enddo
```

(c) Iteration space tiling

```
do JJJ = 1, N, B_JJJ
  do KKK = 1, N, B_KKK
     do II = 1, N, B_II
        do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
           do K = KKK, KKK+B_KKK-1
              do I = II, II+B_II-1
                 do J = JJ, JJ+B_JJ-1
                    if (K.eq.1) C(I,J) = 0
                    C(I,J) = C(I,J) + A(I,K) * D(K,J)
 . . .
enddo
```

(d) Index set splitting

```
do JJJ = 1, N, B_JJJ
  do KKK = 1, N, B_KKK
     do II = 1, N, B_II
        do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
           do K = KKK, min (1, KKK+B_KKK-1)
              do I = II, II+B_II-1
                 do J = JJ, JJ+B_JJ-1
                    C(I,J) = 0
                    C(I,J) = C(I,J) + A(I,K) * D(K,J)
              . . .
           enddo
           do K = max (2, KKK), KKK+B_KKK-1
              do I = II, II+B_II-1
                 do J = JJ, JJ+B_JJ-1
                    C(I,J) = C(I,J) + A(I,K) * D(K,J)
              . . .
           enddo
        . . .
        enddo
```

Figure 2.20: (a) Code of the matrix multiplication with initialization of matrix C. (b) Loop nest after the code sinking transformation, (c) after the iteration space tiling phase and (d) after applying index set to eliminate the if-statements.

nested, we obtain the code of Fig. 2.20b. Now the iteration space can be tiled into different levels of tiles, obtaining the code of Fig. 2.20c. Finally, index set splitting is applied to eliminate the conditional statement of the loop body, obtaining the code of Fig. 2.20d; now, the inner loops (I and J) inside the K loops can be fully unrolled.

2.4 NON-RECTANGULAR ITERATION SPACES

This section summarizes the assumptions on which our compiler strategies, that will be presented in the following chapters, are based.

First of all, we note that the codes for which loop tiling has had the greatest success so far are the so-called numerical or scientific codes that spend most of their time executing nested loops that manipulate matrices of numerical values. Thus, from now on we will only deal with this kind of codes. More precisely, this thesis focuses on the loop tiling transformation applied on loop nests that define *non-rectangular* iteration spaces. Thus, from now on, we also assume that the bounds of the loops in the nest are max/min functions of affine functions of the surrounding loops iteration variables. As an example, Fig. 2.21 shows the code for multiplying a lower triangular matrix (A) by an upper triangular matrix (D). The result is a square matrix (C). This loop nest defines a *non-rectangular* iteration space.

These non-rectangular loop nests commonly appear in numerical applications and current compilers are not capable of restructuring them into multilevel tiled loop nests. Our goal in this thesis is to show that multilevel tiling can also be applied to *non-rectangular* iteration spaces to achieve high performance on modern microprocessors. In this work we mainly deal with the *code updating phase* of compilers when tiling is applied at several levels of the memory hierarchy, and specially, at the register level. Thus, we assume that previous dependence analysis to prove the legality of tiling has already been performed.

Since loop tiling can only be applied to fully permutable and perfectly nested loop nests, we assume that the original loop nest is fully permutable and perfectly nested. However, recall that (a) non fully permutable loop nests can be turned into fully permutable nests by applying a loop skewing

```
do I = 1, N
  do J = 1, N
     do K = 1, min(I,J)
        C(I,J) = C(I,J) + A(I,K) * D(K,J)
     enddo
  enddo
enddo
```

Figure 2.21: Example of loop nest that describes a non-rectangular iteration space. The code corresponds to the multiplication of a lower and an upper triangular matrices.

transformation [121] and (b) non perfectly nested loops can be converted into perfectly nested loops using a code sinking transformation [129]. As mentioned before, to avoid the execution time overhead introduced by the code sinking transformation, it must be undone after tiling.

There has been much work in the literature regarding loop tiling that has been mostly focused on the locality analysis to exploit data reuse at the cache level and less attention has been paid to the register level. In this work, we also deal with the *locality analysis phase* for the register level and we assume that the locality analysis for the other memory hierarchy levels 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.

2.5 SUMMARY

In this chapter, we have given some background on the loop tiling transformation. First, we illustrated the loop tiling transformation with a typical example and showed how tiling can improve data locality. Thereafter, we showed how iterations in perfectly nested loops can be modeled as nodes in an iteration space where the data dependences are the directed arcs between the nodes, representing ordering constraints. We also summarized the theory of unimodular transformations proposed by [121], introducing their legality constraints and different approaches to generate code using these transformations.

Later, we have discussed the three challenges that confront a compiler to perform loop tiling. Figure 2.22 shows the whole sequence of compiler phases for performing loop tiling.

The first challenge is the dependence analysis for determining whether loop tiling is a legal transformation or not. We enunciated the legality of this transformation, given the general dependence framework, and presented an example showing how the loop skewing transformation can be used to enable loop tiling when data dependences in the original code prevent it. We also discussed the significance of having dependence tests with high precision in optimizing compilers for avoiding false dependences that can prevent the compiler from applying loop tiling.

The second challenge is the locality analysis for determining the tiling parameters that deliver the best performance. To guide this analysis, it is necessary to obtain reuse information of the loop

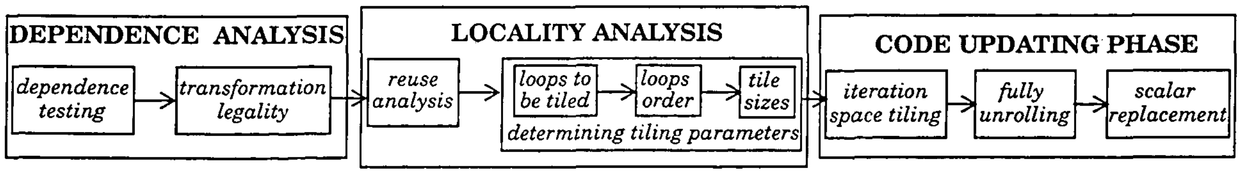


Figure 2.22: Whole sequence of compiler phases for performing loop tiling.

nest. We explained the concept of *reuse vector space* [121] which captures inherent reuse within a loop nest. We also reviewed previous work related to determining the tiling parameters for best cache performance. In order to achieve the best performance, it is important to apply the locality analysis to all the relevant levels of the memory hierarchy simultaneously. We summarized previous works focused on studying the interaction between the different memory levels.

Finally, the third challenge is the *code updating phase* to generate the transformed multilevel tiled code. We showed how conventional techniques implement loop tiling for any memory level, including the register file. Loop tiling is implemented by combining two transformations, namely strip-mining and loop permutation. Moreover, when tiling for enhancing register reuse, an extra phase in needed. In this case, after tiling the iteration space it is necessary to fully unroll the loops that traverse the register tiles. In addition, scalar replacement should be finally applied to expose reuse in the innermost loop. We have also showed how loop tiling can be applied to non-perfectly nested loops by using the code sinking transformation.

We concluded this chapter by summarizing the main assumptions on which our compiler strategies, that will be presented in the following chapters, are based on.

3
TILING FOR THE REGISTER LEVEL

Summary

Tiling is a well-known loop transformation generally used to expose coarse-grain parallelism and to exploit data reuse at the cache level. Tiling can also be used to exploit data reuse at the register level and to improve a programs' ILP. However, previous proposals in the literature (as well as commercial compilers) are only able to perform multi-dimensional tiling for the register level when the iteration space is rectangular. In this chapter we present a new general algorithm to perform multi-dimensional tiling for the register level in both rectangular and non-rectangular iteration spaces. We also propose a simple heuristic to determine the tiling parameters at this level. Finally, we evaluate our method using as benchmarks typical linear algebra algorithms having non-rectangular iteration spaces and compare our proposal against commercial compilers and preprocessors able to perform optimizing code transformations such as inner unrolling, outer unrolling and software pipelining.

3.1 INTRODUCTION

The ability 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. The typical bandwidth provided between the register file and the functional units in current superscalar microprocessors is three or four times higher than the bandwidth provided by the first level cache. Current superscalar microprocessors provide at least 6 floating point register file ports compared to the one or two ports provided by the first level cache. If the register level is not properly exploited, then the number of ports of the first level cache bounds processor performance. Much research has been devoted to exploit data reuse at the cache level [18][27][28][50][69][74][83][90][97][122] while, surprisingly, little attention has been paid to the register level [21][23].

Moreover, modern microprocessors issue multiple instructions per cycle to exploit instruction level parallelism (ILP). To achieve high performance in these processors, compilers not only have to deal with data reuse but they must also extract ILP from the sequential version of the program to exploit the multi-issue capabilities of the hardware. Some compiler transformations such as unrolling [129] and software pipelining [79][106] have been proposed to improve ILP but, unfortunately, do not properly exploit data reuse at the register level.

Loop Tiling [98][110][128] is a transformation with the desirable property that can fulfill both goals at the same time: it exploits data reuse and, if applied to the register level, it can also extract ILP. So far, loop tiling has been basically used to expose coarse-grain parallelism [69][123] and to exploit data reuse at the cache level [27][46][65][122], but has seldom been applied to the register level.

The current state-of-the-art regarding tiling for the register level (see [121][124] and also current commercial compilers), can only perform multi-dimensional tiling for the register level when the iteration space is rectangular. Tiling non-rectangular iteration spaces is not generally considered because it is far from being trivial. In non-rectangular spaces, the bounds of the loops that result from the iteration space tiling do not determine a constant number of iterations, and therefore, the loops that provide reuse at the register level cannot be directly fully unrolled. Recall that fully unrolling is necessary since registers are only addressable by means of absolute addresses (register number).

In this chapter we focus on how multi-dimensional tiling can be applied to the register level for any iteration space having loop bounds defined as compositions (max or min) of affine functions of the surrounding loops iteration variables. This includes those iteration spaces which bounds are not parallel to the iteration space axes, that is, non-rectangular iteration spaces. These non-rectangular iteration spaces are commonly found in linear algebra algorithms or can arise as a result of applying previous transformations such as loop skewing.

Our method applies index set splitting to a previously tiled iteration space to divide it into partitions, such that, in one of these partitions, all loops that provide data reuse at the register level can be fully unrolled. The goal is to obtain one partition where all loops that provide data reuse at the register level perform a constant number of iterations. These loops can then be fully unrolled. After unrolling, we apply scalar replacement [20][23] to remove redundant loads and stores.

Before performing any transformations to better exploit the register level, it is always necessary to perform a locality analysis. This locality analysis will determine which loops are good candidates to be tiled, the tile sizes in each dimension and the arrangement of the loops that deliver the best performance. In this chapter we will also propose a very simple heuristic to determine the tiling parameters at the register level. We will evaluate this heuristic and present results that show the heuristic behaves well in typical linear algebra programs.

After presenting our algorithm and its associated locality analysis heuristic, we evaluate the performance obtained by our method, using 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 such as inner unrolling, outer unrolling and software pipelining. Measurements were taken on two different superscalar microprocessors, an ALPHA 21164 and a MIPS R10000.

Finally, we want to note that *unroll-and-jam* [23][26] is also known as *register tiling* in the literature, because it also rearranges the iteration-space traversal of a loop nest, so that it results in a series of small polyhedra executed one after the other. Indeed, for rectangular iteration spaces, unroll-and-jam applied to several loops in a nest is the same transformation as register tiling. However, for non-rectangular iteration spaces, unroll-and-jam and register tiling do not generate the same code. Moreover, previous proposals of unroll-and-jam can only be applied to limited cases of non-rectangular iteration spaces [23]. In this chapter we will discuss in depth the conceptual differences between *unroll-and-jam* and register tiling and we will evaluate the performance obtained from each technique.

The remainder of this chapter is organized as follows: Section 3.2 presents our general method to perform tiling at the register level. In Section 3.3 the benefits of tiling for the register level are explained and in Section 3.4 we propose a simple heuristic that determines the tiling parameters at the register level. Section 3.5 presents the evaluation of our method through some experimental results. Finally, Section 3.6 presents the previous work related to register tiling and in Section 3.7 we summarize this chapter.

3.2 IMPLEMENTATION OF REGISTER TILING

In this section we will describe our method to perform tiling for the register level and we will also measure its complexity. The method consists in a combination of well-known transformations: strip-mining, loop permutation, index set splitting, unrolling and scalar replacement.

As mentioned in chapter 2, 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. In this particular section, 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. We will deal with the locality analysis at the register level later in Section 3.4.

3.2.1 Overview

Our method divides first the iteration space defined by the loop structures into regular tiles using the strip-mining and loop permutation transformations [55][59][123]. Strip-mining is used to partition the iteration space and loop permutation 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). After the iteration space tiling phase, we obtain a new single loop nest that traverses all tiles that cover the entire original iteration space.

To exploit the register level it is now necessary to fully unroll the loops that step the points inside the register tiles (the *element* loops). Fully unrolling is necessary because registers, as opposed to cache memory locations, are only addressable using absolute addresses (register number). A necessary condition for a loop to be fully unrollable is that it must always execute exactly the same number of iterations. Unfortunately, after the iteration space tiling phase, the element loops do not fulfill this condition, i.e., they do not always execute the same number of iterations.

As an example, Fig. 3.1 shows two iteration spaces, one rectangular and one non-rectangular, after the iteration space tiling phase has been applied. In the figure, each register tile has been grouped within a box. Note that there are two types of tiles: *boundary*-tiles (grey-shaded boxes) and *core*-tiles (white-shaded boxes). We define a *boundary*-tile as a tile whose intersection with the original iteration space is not equal to the tile, and we define a *core*-tile as a tile whose intersection with the original iteration space is equal to the tile. When the element loops are traversing *core*-tiles, they always execute as many iterations as the tile size, while when the element loops are traversing *boundary*-tiles, the number of iterations is less or equal than the tile size. Therefore, the problem is that the element loops do not always execute the same number of iterations.

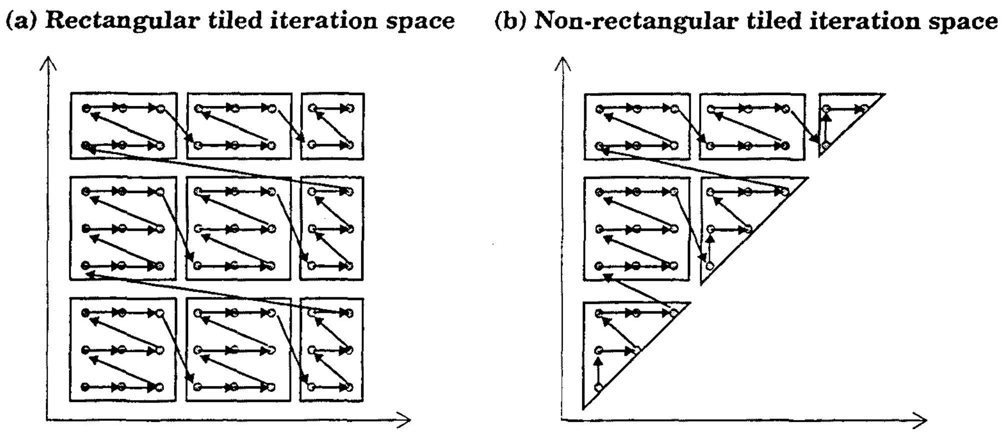

(a) Rectangular tiled iteration space
(b) Non-rectangular tiled iteration space

Figure 3.1: Example of (a) rectangular and (b) non-rectangular iteration space.

This problem can be solved by taking the single loop nest resulting from the iteration space tiling transformation and breaking it into several loop nests, one of which will traverse all (and only) *core*-tiles while the remaining loop nests will traverse the different types of *boundary*-tiles.

In rectangular iteration spaces, it is very easy to break the original, tiled, loop nest as just described because the intersection of a *boundary*-tile with the original iteration space is always a rectangular space. As an example, consider the rectangular iteration space defined by the loop nest of Fig. 3.2a. After tiling the iteration space in both dimensions, we obtain the loop nest of Fig. 3.2b that traverses the entire iteration space as shown in Fig. 3.2c. Since all the *boundary*-tiles are rectangular, it is possible to create a loop nest that traverses all *core*-tiles by simply adjusting the outer tile loops bounds, as shown in Fig. 3.2d.

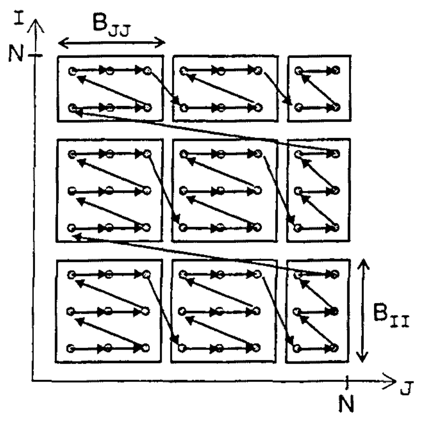
(a) Original loop nest

```
    do I = 1, N
      do J = 1, N
        loop body
    enddo
```

(b) Tiled loop nest

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    do I = II, min(N, II+B_II-1)
      do J = JJ, min(N, JJ+B_JJ-1)
        loop body
enddo
```

(c) Tiled iteration-space traversal



(d) Tiled loop nest

```
do II = 1, N-B_II+1, B_II
  do JJ = 1, N-B_JJ+1, B_JJ
    do I = II, II+B_II-1
      do J = JJ, JJ+B_JJ-1
        loop body
  enddo
```

```
  do I = II, II+B_II-1
    do J = JJ, N
      loop body
enddo
do JJ = 1, N, B_JJ
  do I = II, N
    do J = JJ, min(N, JJ+B_JJ-1)
      loop body
enddo
```

Figure 3.2: (a) Example of loop nest that describes a rectangular iteration space. (b) Loop nest after tiling the iteration space in both dimensions. (c) Order in which iterations are executed in the tiled iteration space. (d) Tiled loop nest after creating the loop nest that traverses all core-tiles.

(a) Original loop nest

```
do I = 1, N
   do J = 1, I
      loop body
enddo
```

(b) Tiled loop nest

```
do II = 1, N, $B_{II}$
   do JJ = 1, min(N, II+$B_{II}$-1), $B_{JJ}$
      do I = II, min(N, II+$B_{II}$-1)
         do J = JJ, min(I, JJ+$B_{JJ}$-1)
            loop body
enddo
```

(c) Tiled iteration-space traversal

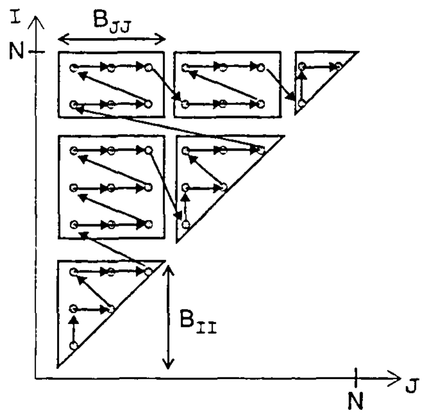


Figure 3.3: (a) Example of loop nest that describes a non-rectangular iteration space. (b) Loop nest after tiling the iteration space in both dimensions. (c) Order in which iterations are executed in the tiled iteration space.

By contrast, in non-rectangular iteration spaces the *boundary*-tiles might be non-rectangular because the element loops have bounds that are affine functions of other element loop iteration variables. Figure 3.3a shows a loop nest that defines a non-rectangular iteration space. After tiling the iteration space in both dimensions, we obtain the loop nest of Fig. 3.3b that traverses the entire iteration space as shown in Fig. 3.3c. Note that in this case, breaking the loop nest in Fig. 3.3b into different loop nests that traverse core-tiles and boundary-tiles is not as trivial as it was in the rectangular iteration space case. Here, element loop J has a bound that depends on element loop I, and, therefore, adjusting the bounds of loops II and JJ is not enough to achieve a loop nest that traverses all (and only) *core*-tiles. The rest of this section will propose our strategy to tackle this problem by using index set splitting [129].

Once the original tiled loop nest has been broken into several loop nests, we proceed to fully unroll the element loops that traverse the *core*-tiles. Fully unrolling is achieved 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. After this process, a new loop body is obtained.

At last, after fully unrolling the element loops, scalar replacement [20] [23] is used to eliminate redundant loads and stores in the loop body.

Summarizing, our strategy performs the following transformations:

- Strip-mining and loop permutation to tile the iteration space into regular tiles,
- index set splitting to be able to fully unroll the element loops,
- fully unrolling, and
- scalar replacement to eliminate unnecessary loads/stores.

We note that our strategy 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, and all other steps remain unchanged.

In the remainder of this section, 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 and, finally, we give analytical expressions both for the complexity of our method and for the amount of code generated.

3.2.2 A Step-by-step Example of Register Tiling

To see how our method works, we will show all the steps we perform in order to apply register tiling to the triangular matrix product algorithm, shown in Fig. 3.4a, which has a non-rectangular 3-dimensional iteration space. We assume that tiling is being applied only for the register level and we suppose that the locality analysis has already been performed. As we will see later in Section 3.4, the locality analysis will always tile *(n-1)* dimensions of a *n*-dimensional iteration space. In our example, we assume that the loops to be tiled are loops J and I and the tile sizes are B_{JJ} and B_{II}, respectively. For simplicity, we suppose that N is multiple of B_{JJ} and B_{II}.

Iteration Space Tiling

The first step of our method consists in dividing the iteration space defined by the original loop nest into regular tiles. This can be done by combining the strip-mining and loop permutation transformations as explained in Chapter 2. The code after tiling the iteration space is shown in Fig. 3.4b. We will refer to the loops that we want to unroll as Unroll Candidate Loops (UCLs). In Fig. 3.4b, the UCLs are loops J and I.

In the next step of our method, index set splitting is used to isolate a partition were the UCLs iterate exactly as many times as the tile size in their dimension. As we discussed in the previous subsection, this particular partition is the partition where the UCLs can be fully unrolled.

(a) Original loop nest

```
do K = 1, N
  do I = K, N
    do J = K, N
      C(I,J) = C(I,J) + A(I,K) * D(K,J)
enddo
```

$\xrightarrow[\text{tiling}]{\text{iteration space}}$

(b) Tiled loop nest

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    do K = 1, min(JJ+B_JJ-1, II+B_II-1)
      do J = max(JJ, K), JJ+B_JJ-1
        do I = max(II, K), II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
enddo
```

Figure 3.4: (a) Original code (form KIJ) of triangular matrix product.
(b) Code after tiling for the register level.

Index Set Splitting

Let $\texttt{II}$ be the iteration variable of a *tile* loop and let B_{II} be the tile size in its dimension. Its associated *element* loop (that is, an UCL) always has $\texttt{II}$ and $\texttt{II}+B_{II}-1$ 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* bounds of the UCL. An *effective* bound is the bound component that results after evaluating a composition (max or min) of bound components. If we achieve this goal, we will be able to fully unroll the UCLs in this partition.

Index Set Splitting (ISS) is a code transformation that 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 UCLs) with the goal of dividing the tiled iteration space into the desired partitions.

To see how we perform ISS we will continue with the example of Fig. 3.4. The bounds of the UCLs $\texttt{J}$ and $\texttt{I}$ in the tiled code of our example are:

$$\max(JJ, K) \leq J \leq JJ+B_{JJ}-1 \qquad \text{and} \qquad \max(II, K) \leq I \leq II+B_{II}-1$$

and our goal is to achieve a loop nest where loops $\texttt{J}$ and $\texttt{I}$ have the following bounds:

$$JJ \leq J \leq JJ+B_{JJ}-1 \qquad \text{and} \qquad II \leq I \leq II+B_{II}-1$$

The loop bounds of the UCLs in the tiled code determine the set of all conditions that must hold in the partition where the UCLs can be fully unrolled. In our example, UCLs $\texttt{J}$ and $\texttt{I}$ could be fully unrolled if the conditions $JJ \geq K$ and $II \geq K$ did hold. Thus, we will apply ISS with each of these conditions.

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

Figure 3.5a 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 3.5b shows the code after applying ISS to loop $\texttt{K}$ and in Fig. 3.5c we can see graphically how the iteration space defined by loops $\texttt{I}$ and $\texttt{K}$ is split.

(a) Before Index Set Splitting

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    do K = 1, min(JJ+B_JJ-1, II+B_II-1)
      do J = max(JJ, K), JJ+B_JJ-1
        do I = max(II, K), II+B_II-1
          C(I,J) = C(I,J) + A(I,K) *D(K,J)
        enddo
      enddo
    enddo
  enddo
enddo
```

index set splitting with restriction K ≤ II $\longrightarrow$

(b) After Index Set Splitting

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    do K = 1, min(II, JJ+B_JJ-1, II+B_II-1)
      do J = max(JJ, K), JJ+B_JJ-1
        do I = II, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo                                   partition 1
      enddo                                       K≤II
    enddo
    do K = max(1,II+1), min(JJ+B_JJ-1, II+B_II-1)
      do J = max(JJ, K), JJ+B_JJ-1
        do I = K, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo                                   partition 2
      enddo                                       K>II
    enddo
  enddo
enddo
```

(c) Iteration space defined by I and K

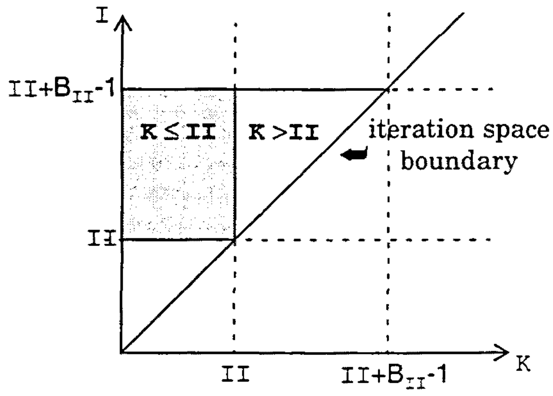


Figure 3.5: a) Loop nest before applying ISS. b) Loop nest after applying ISS with restriction K ≤ II to loop K. c) Iteration space defined by loops I and K.

After applying ISS to loop K, the bounds of loop I in both partitions can be simplified. In the partition where K ≤ II holds (partition 1), the lower bound of I can be simplified to II (max (II, K)= II). In a similar way, in the partition where K ≤ II never holds (partition 2), the lower bound of I can be simplified to K (max (II, K)= K). In Fig. 3.5b the main changes on the loop bounds are marked in bold.

Now loop I of partition 1 always executes B_{II} iterations and can be fully unrolled. Loop I of the partition 2 never executes a constant number of iterations and cannot be fully unrolled. We say that loop I of partition 2 is no longer an UCL.

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. 3.5b we have to apply ISS again to partition 1 with the condition JJ ≥ 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. 3.5b, for example, loop J can be fully unrolled if ISS is applied again to this partition. Our method also deals with all these partitions. In these partitions, it is sometimes necessary to perform a loop permutation to make sure the UCLs become innermost loops before fully unrolling them.

Figure 3.6a shows the final code of our example after applying ISS repeatedly to all partitions. The first loop nest (partition 1) traverses the partition containing only *core*-tiles and, therefore, loops J and I can be fully unrolled (they always execute B_{JJ} and B_{II} 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). Note also that in the third loop nest a loop permutation has been performed to make loop J become the innermost loop in the nest.

Unrolling and Scalar Replacement

In our final steps, we fully unroll the loops and apply scalar replacement to eliminate redundant loads and stores in each partition (see Chapter 2, Section 2.3.3). The final resulting code after all steps is shown in Fig. 3.6b (assuming B_{JJ}=B_{II}=2).

3.2.3 Index Set Splitting Algorithm

In this section we formalize how to apply index set splitting repeatedly. We will first present how to build all the conditions that must hold in the partition where all UCLs can be fully unrolled. These conditions determine the restrictions used to split the loops. We then present how a loop is split given one restriction. Afterwards we present the order in which we split the loops. This order is very important to avoid code explosion and to avoid processing a loop more than once. Finally, we give the complete ISS algorithm.

All discussion in this section assumes that the iteration space tiling phase has already been performed. After tiling the iteration space into regular tiles we obtain an m-deep loop nest with the following structure:

```
do $I_1 = L_1, U_1, B_1$           ⎤
                                   ⎥ tile loops
  do $I_2 = L_2, U_2, B_2$         ⎦
    . . .                          ] non-tiled loop
      do $I_m = L_m, U_m, 1$       ] UCLs
        . . .
enddo
```

where I_1 is the most external loop and I_m is the most internal one.

(a) After the Index Set Splitting process

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    do K = 1, min(II, JJ)
      do J = JJ, JJ+B_JJ-1
        do I = II, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo
      enddo
    enddo                                   partition 1
    do K = max(1, JJ+1), min(JJ+B_JJ-1, II)
      do J = K, JJ+B_JJ-1
        do I = II, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo
      enddo
    enddo                                   partition 2
    do K = max(1, II+1), min(JJ, II+B_II-1)
      do I = K, II+B_II-1
        do J = JJ, JJ+B_JJ-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo
      enddo
    enddo                                   partition 3
    do K = max(1, II+1, JJ+1), min(JJ+B_JJ-1, II+B_II-1)
      do J = K, JJ+B_JJ-1
        do I = K, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
        enddo
      enddo
    enddo                                   partition 4
  enddo
enddo
```

(b) After fully unrolling and applying scalar replacement

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do K = 1, min(II, JJ)
      RR1 = RR1 + A(II,K) * D(K,JJ)
      RR2 = RR2 + A(II,K) * D(K,JJ+1)
      RR3 = RR3 + A(II+1,K) * D(K,JJ)
      RR3 = RR3 + A(II+1,K) * D(K,JJ+1)
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
    do K = max(1, JJ+1), min(JJ+B_JJ-1, II)
      RR1 = A(II,K)
      RR2 = A(II+1,K)
      do J = K, JJ+B_JJ-1
        C(II,J) = C(II,J) + RR1 * D(K,J)
        C(II+1,J) = C(II+1,J) + RR2 * D(K,J)
      enddo
    enddo
    do K = max(1, II+1), min(JJ, II+B_II-1)
      RR1 = D(K,JJ)
      RR2 = D(K,JJ+1)
      do I = K, II+B_II-1
        C(I,JJ) = C(I,JJ) + A(I,K) * RR1
        C(I,JJ+1) = C(I,JJ+1) + A(I,K) * RR2
      enddo
    enddo
    do K = max(1, II+1, JJ+1), min(JJ+B_JJ-1, II+B_II-1
      do J = K, JJ+B_JJ-1
        RR1 = D(K,J)
        do I = K, II+B_II-1
          C(I,J) = C(I,J) + A(I,K) * RR1
        enddo
      enddo
    enddo
  enddo
enddo
```

Figure 3.6: (a) Code after applying index set splitting repeatedly. (b) Code after fully unrolling and applying scalar replacement ($B_{II}=B_{JJ}=2$).

The UCLs are always the most internal loops and their steps are always 1, and the tile loops are always the most external loops and their steps are the tile sizes in their dimensions. As already mentioned, we will see later in Section 3.4 that the locality analysis will always tile (*n-1*) dimensions of a *n*-dimensional iteration space and the non-tiled loop will be placed between the tile loops and the UCLs.

The bounds of the loops in the tiled code have the form:

$$\mathsf{L}_i = max\left(l_{i,0}, l_{i,1}, \ldots, l_{i,r_i}\right) \qquad \mathsf{U}_i = min\left(u_{i,0}, u_{i,1}, \ldots, u_{i,s_i}\right)$$

The first subscript of $l_{i,q}$ and $u_{i,q}$ identifies the loop and the second subscript enumerates the loop bound components of the loop. Note that one loop $\mathtt{I}_i$ has $r_i + 1$ lower bound components and $s_i + 1$ upper bound components.

Each bound component $l_{i,q}$ ($u_{i,q}$) is a ceiling (floor) function of an affine function of the surrounding loops iteration variables. The ceiling and floor functions appear in the tiled code due to the strip mining and loop permutation transformations used in the iteration space tiling phase[1]. Each bound component $l_{i,q}$, $u_{i,q}$ can be expressed as follows:

$$l_{i,q} = \left\lceil \frac{\sum_{k=1}^{i-1} a_{i,q}^k \cdot \mathtt{I}_k + \Upsilon_{i,q}}{a_{i,q}^i} \right\rceil \qquad u_{i,q} = \left\lfloor \frac{\sum_{k=1}^{i-1} b_{i,q}^k \cdot \mathtt{I}_k + \varphi_{i,q}}{b_{i,q}^i} \right\rfloor$$

where $a_{i,q}^k, b_{i,q}^k \in Z \ (1 \le k \le i)$, and $\Upsilon_{i,q}$ and $\varphi_{i,q}$ can be constants or parameters. We call L_i and U_i the lower and upper bounds of loop $\mathtt{I}_i$, respectively, and we refer to each bound component $l_{i,q}$ ($u_{i,q}$) as a lower (upper) *simple bound.*

Conditions

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 ($\mathtt{I}_j$) in the lower bound and the iteration variable of the *tile* loop plus the tile size minus one ($\mathtt{I}_j + \mathsf{B}_j - 1$) 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_i 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.

1.A floor function can be converted in a ceiling function in the following way: $\left\lfloor \frac{a}{b} \right\rfloor = \left\lceil \frac{a-b+1}{b} \right\rceil$

Let I_i be an UCL, whose corresponding tile loop is I_j ($j < i$) and the tile size in its dimension is B_j. Then, the bounds of UCL I_i have the following form:

$$\mathrm{L}_i = max\left(\mathrm{I}_j, l_{i,1}, \ldots, l_{i,r_i}\right) \qquad \mathrm{U}_i = min\left(\mathrm{I}_j + \mathrm{B}_j - 1, u_{i,1}, \ldots, u_{i,s_i}\right)$$

and the set of conditions C_i determined by this UCL is:

$$C_i = \{\mathrm{I}_j \geq l_{i,q_1}, \mathrm{I}_j + \mathrm{B}_j - 1 \leq u_{i,q_2} | (1 \leq q_1 \leq r_i, 1 \leq q_2 \leq s_i)\}$$

From now on, we will distinguish two kinds of conditions: conditions generated by lower bound components of loop I_i and conditions generated by upper bound components of loop I_i. The conditions generated by lower bound components of loop I_i have the form $\mathrm{I}_j \geq l_{i,q}$ and we will use the notation $C^l_{i,q}$ $(1 \leq q \leq r_i)$ to refer to the condition generated by loop I_i, when comparing the lower simple bound I_j with the lower simple bound $l_{i,q}$. On the other hand, the conditions generated by upper bound components of loop I_i have the form $\mathrm{I}_j + \mathrm{B}_j - 1 \leq u_{i,q}$ and we will use the notation $C^u_{i,q}$ $(1 \leq q \leq s_i)$ to refer to the condition generated by loop I_i, when comparing the upper simple bound $\mathrm{I}_j + \mathrm{B}_j - 1$ with the upper simple bound $u_{i,q}$.

Let m be the total number of loops in the loop nest after tiling and let w be the number of UCLs. The w UCLs of the m total loops are always in the innermost positions after tiling the iteration space; then, the set C of all conditions is:

$$C = \bigcup_{i=n-w+1}^{n} \{C_i\} = \bigcup_{i=n-w+1}^{n} \{\{C^l_{i,q} | 1 \leq q \leq r_i\} \cup \{C^u_{i,q} | 1 \leq q \leq s_i\}\}$$

and the total number of conditions we have to deal with initially is:

$$|C| = \sum_{i=n-w+1}^{n} (r_i + s_i)$$

Restrictions

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's now see how the loop to be split is identified and how index set splitting is applied to this loop.

Let I_i be again the UCL that generates a particular condition of the type $C^l_{i,q}$ (that is, a condition generated by the lower bound of I_i), and let I_j ($j < i$) and B_j be the corresponding tile loop and tile size, respectively.

The condition $C^l_{i,q}$ can be written into the form:

$$\sum_{k=1}^{i-1} g^k_{i,q} \cdot \mathrm{I}_k + \Gamma_{i,q} \leq 0 \qquad (1)$$

where $g^k_{i,q} = a^k_{i,q}$ (for $1 \leq k \leq i-1$ and $k \neq j$), $g^k_{i,q} = a^k_{i,q} - a^i_{i,q}$ (for $k = j$) and $\Gamma_{i,q} = \Upsilon_{i,q}$.

The most internal loop that appears in the expression (1) is identified by the last non-zero coefficient $g^k_{i,q}$ with $1 \le k \le i-1$. This loop is the loop to be split and it can be an UCL or not.

Let I_p be the loop to be split ($p \le i-1$). We obtain the *restriction* used to perform ISS by solving the condition (expression (1)) for I_p. Two types of restrictions can be obtained depending on the sign of $g^p_{i,q}$ (the coefficient of I_p): when the coefficient is positive ($g^p_{i,q} > 0$) we have an *upper restriction* that has the following form:

$$I_p \le u'_{i,q} \qquad \text{, where } u'_{i,q} = \left\lfloor \frac{\sum_{k=1}^{p-1} -g^k_{i,q} \cdot I_k - \Gamma_{i,q}}{g^p_{i,q}} \right\rfloor \qquad (2)$$

and when the coefficient is negative ($g^p_{i,q} < 0$) we have a *lower restriction* with the following form:

$$I_p \ge l'_{i,q} \qquad \text{, where } l'_{i,q} = \left\lceil \frac{\sum_{k=1}^{p-1} g^k_{i,q} \cdot I_k + \Gamma_{i,q}}{-g^p_{i,q}} \right\rceil \qquad (3)$$

We have developed the expressions using a condition of the type $C^l_{i,q}$, but a similar analysis can be done for a condition of the type $C^u_{i,q}$ (that is, a condition generated by the upper bound of I_i). A condition of the type $C^u_{i,q}$ can also be written as shown in expression (1), but in this case $g^k_{i,q} = -b^k_{i,q}$ (for $1 \le k \le i-1$ and $k \ne j$), $g^k_{i,q} = b^i_{i,q} - b^k_{i,q}$ (for $k = j$) and $\Gamma_{i,q} = b^i_{i,q} \cdot B_j - b^i_{i,q} - \varphi_{i,q}$. The rest of the analysis is the same as before.

Thus, from each kind of condition ($C^l_{i,q}$ or $C^u_{i,q}$) we can obtain both types of restriction (see Table 3.1). Now we will explain how index set splitting is applied with each type of restriction. The loop to be split (loop I_p), according to a particular restriction, is divided into two consecutive loops that iterate over non-intersecting partitions of the iteration space: in one partition the restriction holds and in the other the restriction does not hold. The type of the restriction determines in which of the two partitions the restriction holds.

	Restriction	
Condition	UPPER ($g^p_{i,q} > 0$)	LOWER ($g^p_{i,q} < 0$)
$C^l_{i,q}$ ($I_j \ge l_{i,q}$)	$I_p \le u'_{i,q}$	$I_p \ge l'_{i,q}$
$C^u_{i,q}$ ($I_j + B_j - 1 \le u_{i,q}$)	$I_p \le u'_{i,q}$	$I_p \ge l'_{i,q}$

Table 3.1: Types of restrictions obtained by each kind of conditions.

Upper Restriction

Let I_p be a loop to be split according to an upper restriction ($I_p \le u'_{i,q}$). Recall that I_i is the UCL that has generated the condition and I_j is its corresponding tile loop. Before applying index set splitting, loop I_p has the following form:

```
do $I_p = L_p, U_p, B_p$
  ...
    do $I_i = max(I_j, l_{i,1}, \dots, l_{i,r_i}), min(I_j + B_j - 1, u_{i,1}, \dots, u_{i,s_i}), 1$
      ...
enddo
```

We want to split loop I_p in such a way that in one of the two partitions the restriction $I_p \le u'_{i,q}$ holds, and in the other partition the restriction does not hold. Thus, loop I_p is split in the following manner:

```
do $I_p = L_p, min(U_p, u'_{i,q}), B_p$
  ...                                      partition 1
enddo
do $I_p = I_p, U_p, B_p$
  ...                                      partition 2
enddo
```

In the first partition the restriction $I_p \le u'_{i,q}$ holds (the upper bound of I_p is $min(U_p, u'_{i,q})$). In the second partition, however, it does not hold; note that in the second partition the lower bound of I_p is the last value of the previous I_p loop. Thus, if $u'_{i,q} < U_p$, then I_p of the second partition will be greater than $u'_{i,q}$ and, if $u'_{i,q} \ge U_p$, then the second partition will not be executed.

Note that, if the step of the split loop is equal to one ($B_p = 1$), the second partition can be written as follows:

```
do $I_p = max(L_p, u'_{i,q} + 1), U_p$
  ...
enddo
```

In the case that I_p is an UCL, we will rewrite the lower bound of the second partition this way, because we need that in one component of the lower bound appears the iteration variable of its corresponding tile loop (one component of L_p) to be able to apply ISS later on.

If a restriction holds in a partition, the condition that generated this restriction also holds. Therefore in both partitions there is a component of the lower or upper bound of loop I_i (I_i was the UCL that generated the condition) that is redundant and can be removed.

More precisely, if the restriction $I_p \le u'_{i,q}$ comes from a condition of the form $C^l_{i,q}$ ($I_j \ge l_{i,q}$), in the first partition we can remove the lower bound component $l_{i,q}$ of loop I_i and in the second partition we can remove the lower bound component I_j. After eliminating these bounds the code has the following structure:

```
do $I_p = L_p, min(U_p, u'_{i,q}), B_p$
  ...
    do $I_i = max(I_j, l_{i,1}, ..., l_{i,q-1}, l_{i,q+1}, ..., l_{i,r_i}), U_i, 1$      partition 1.
    ...
enddo
do $I_p = I_p, U_p, B_p$
  ...
    do $I_i = max(l_{i,1}, ..., l_{i,r_i}), U_i, 1$      partition 2
    ...
enddo
```

Likewise, if the upper restriction $I_p \le u'_{i,q}$ comes from a condition of the form $C^u_{i,q}$ ($I_j + B_j - 1 \le u_{i,q}$), in the first partition we can remove the upper bound component $u_{i,q}$ of loop I_i and in the second partition we can remove the upper bound component $I_j + B_j - 1$.

In partition 1, where the restriction holds, loop I_i 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 partition 2, loop I_i 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.

At this point we want to direct the reader's attention to the particular case where the loop to be split (loop I_p) is also an UCL. When we apply ISS to loop I_p, a new upper bound component (namely, $u'_{i,q}$) appears in loop I_p of partition 1. If that loop is an UCL we will need to add a new condition to the set C of conditions. Moreover, in partition 2, where the restriction does not hold, loop I_p is no longer an UCL, because loop I_i (that has generated the condition) has bounds that are affine functions of I_p.

Lower Restriction

Let now I_p be a loop to be split according to a lower restriction ($I_p \geq l'_{i,q}$). We want to split loop I_p in such a way that in one of the two partitions the restriction $I_p \geq l'_{i,q}$ holds, and in the other partition the restriction does not hold. For a lower restriction, loop I_p is split in the following manner:

```
do $I_p = L_p, min(U_p, l'_{i,q} - 1), B_p$    ]
  ...                                        ] partition 1
enddo                                        ]
do $I_p = I_p, U_p, B_p$                      ]
  ...                                        ] partition 2
enddo                                        ]
```

In this case, the restriction $I_p \geq l'_{i,q}$ holds in the second partition and it does not hold in the first partition.

As explained before, if loop I_p is an UCL, we need the iteration variable of its corresponding tile loop to appear in one component of its lower bound. Thus, when I_p is an UCL, we rewrite the lower bound of the second partition as follows:

```
do $I_p = max(L_p, l_{i,q}), U_p$
  ...
enddo
```

Now, in both partitions there is a component of the lower or upper bound of loop I_i that is redundant and can be removed. If the restriction $I_p \geq l'_{i,q}$ comes from a condition of the form $C^l_{i,q}$ ($I_j \geq l_{i,q}$), in the second partition we can remove the lower bound component $l_{i,q}$ of loop I_i and in the first partition we can remove the lower bound component I_j. After eliminating these bounds the code has the following structure:

```
do $I_p = L_p, min(U_p, l'_{i,q} - 1), B_p$                                    ]
  ...                                                                        ]
    do $I_i = max(l_{i,1}, ..., l_{i,r_i}), U_i, 1$                            ] partition 1
      ...                                                                    ]
enddo                                                                        ]
do $I_p = I_p, U_p, B_p$                                                      ]
  ...                                                                        ]
    do $I_i = max(I_j, l_{i,1}, ..., l_{i,q-1}, l_{i,q+1}, ..., l_{i,r_i}), U_i, 1$  ] partition 2
      ...                                                                    ]
enddo                                                                        ]
```

Likewise, if the lower restriction $\mathtt{I}_p \geq l'_{i,q}$ comes from a condition of the form $C^u_{i,q}$ ($\mathtt{I}_j + \mathtt{B}_j - 1 \leq u_{i,q}$), in the second partition we can remove the upper bound component $u_{i,q}$ of loop $\mathtt{I}_i$ and in the first partition we can remove the upper bound component $\mathtt{I}_j + \mathtt{B}_j - 1$. In partition 2, where the restriction holds, loop $\mathtt{I}_i$ is still an UCL and in partition 1 loop $\mathtt{I}_i$ is no longer an UCL.

In the particular case in which the loop to be split (loop $\mathtt{I}_p$) is an UCL, a new lower bound component (namely, $l'_{i,q}$) appears in loop $\mathtt{I}_p$ of partition 2. Thus, we will have to add a new condition to the set C of conditions. Moreover, in partition 1, where the restriction does not hold, loop $\mathtt{I}_p$ is no longer an UCL, because loop $\mathtt{I}_i$ (that has generated the condition) has bounds that are affine functions of $\mathtt{I}_p$.

Processing Order

We have seen so far that the bounds of the UCLs determine the set of conditions that must hold in the partition where all UCLs can be fully unrolled. Each condition determines the restriction used to split the loop. 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 is also very important to reduce code expansion.

If we split the loops from outermost to innermost, we reduce code expansion[2], because this order generates less partitions containing boundary-tiles, that is, partitions where the inner loops cannot be fully unrolled. As an example, consider the code in Fig. 3.7a. Suppose that there are two upper restrictions[3]: one splits loop `I` and the other splits loop `J`. If we split the loops from outermost to innermost (first loop `J` and then loop `I`) we obtain the code shown in Fig. 3.7b, where the partition inside the shaded rectangles is the partition were the applied restrictions hold. It can be seen that loop `I` is only split in one partition of loop `J`, more exactly, in the partition where loop `J` can be fully unrolled. Nevertheless, if we split the loops from innermost to outermost (first loop `I` and then loop `J`) we obtain the code shown in Fig. 3.7c. In this case, it can be seen that loop `I` is split in both partition of loop `J`.

Thus, splitting the loops from outermost to innermost reduces code expansion, because the loop nests (or partitions) where the inner loops cannot be fully unrolled are written in a more compact form.

However, when we apply index set splitting to a loop, a new component appears in the bounds of this loop. If the loop being split is an UCL we will have to deal with a new restriction that will split an outer loop. Therefore, splitting the loops from outermost to innermost would induce the need to split loops that have already been processed, leading to repeated processing of some loops. As an example, consider the code shown in Fig. 3.8a, where loops `I` and `J` are UCLs. Suppose we are dealing with the

2.Every time ISS is applied, the loop body of the loop being split is replicated.
3.The type of the restrictions is indifferent. We use upper restrictions for simplicity in the explanation.

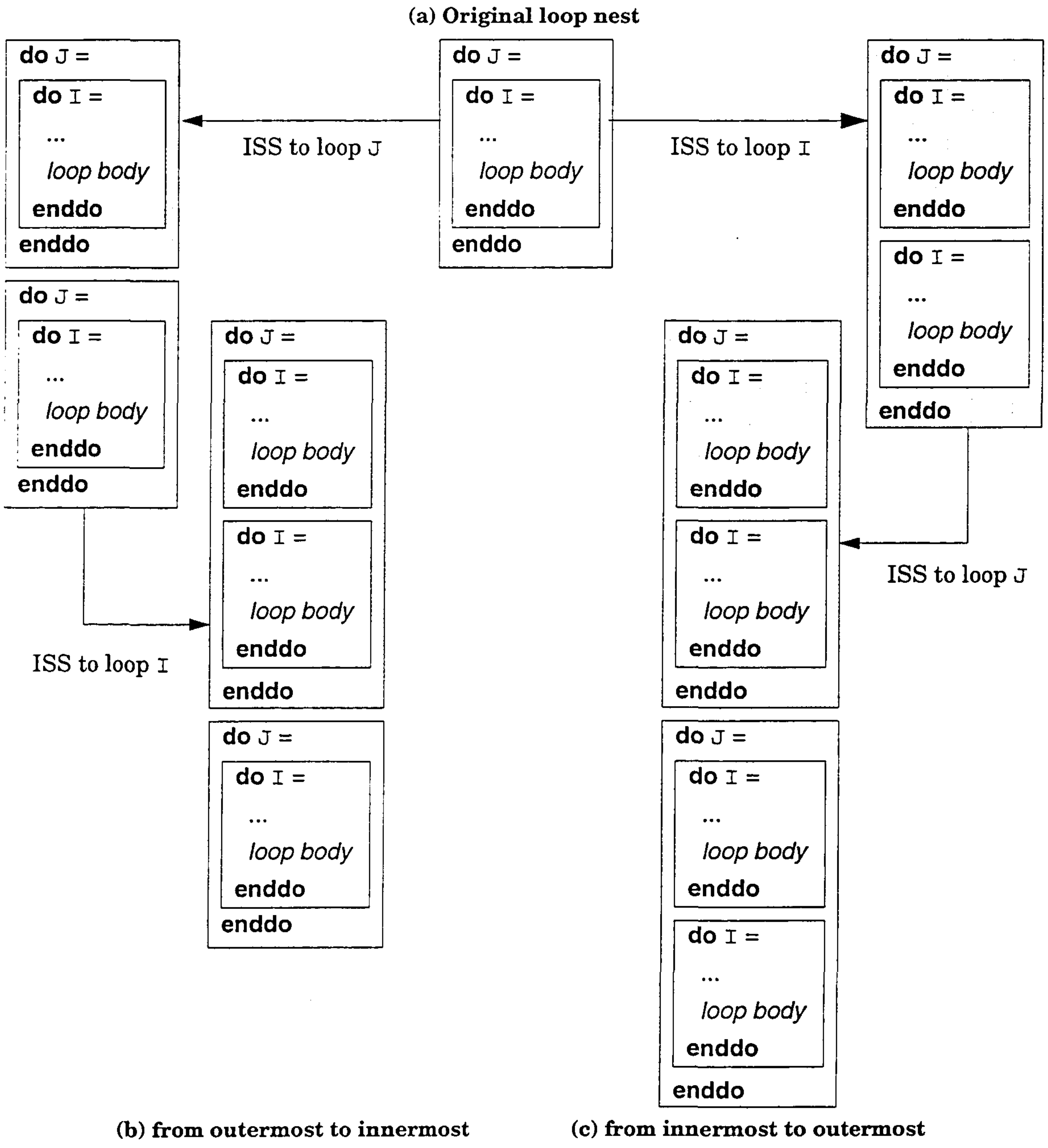


Figure 3.7: (a) Example of loop nest. (b) Loop nests after splitting the loops from outermost to innermost. (c) Loop nests after splitting the loops from innermost to outermost.

(a) Original loop nest

```
do JJ = . . .
  do II = . . .
    . . .
    do J = JJ, JJ+B_JJ-1
      do I = max(J, II), II+B_II-1
        loop body
  . . .
enddo
```

(b) After applying ISS to loop J

```
do JJ = . . .
  do II = . . .
    . . .
    do J = JJ, min(II, JJ+B_JJ-1)
      do I = II, II+B_II-1
        loop body
    enddo
    do J = max(JJ, II+1), JJ+B_JJ-1
      do I = J, II+B_II-1
        loop body
    enddo
  . . .
enddo
```

Figure 3.8: (a) Example of loop nest, where loops I and J are UCLs. (b) Loop nest after applying ISS to loop J, with restriction $J \leq II$. A new upper bound component (namely, II) has appeared in loop J.

loops from outermost to innermost and now we have to deal with the upper restriction $J \leq II$ that splits loop J. After applying ISS, we obtain the code shown in Fig. 3.8b (the restriction holds in the first partition). We can see that UCL J has now a new upper bound component (namely, II). Therefore we have a new restriction ($II \geq JJ+B_{JJ}-1$) to deal with to be able to fully unroll loop J. This new restriction splits loop II, that has already been processed.

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. However, as previously shown in Fig. 3.7c, the problem of this order is that it produces a substantial code expansion.

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 minimizes code expansion.

This order of dealing with the restrictions could make the UCLs not to be directly surrounded by a non-UCL loop. However, to be able to apply scalar replacement later on, it is necessary that all UCLs are perfectly nested and directly surrounded by a non-UCL loop. To achieve this requirement, our algorithm applies loop distribution to the innermost non-UCL loop after the ISS process. As an example, consider the code shown in Fig. 3.9a, where loops I and J are the UCLs. After applying index set splitting repeatedly to be able to fully unroll loops I and J, we obtain the code shown in Fig. 3.9b.

(a) Original loop nest

```
do JJ = $L_{JJ}$, $U_{JJ}$, $B_{JJ}$
  do II = $L_{II}$, $U_{II}$, $B_{II}$
    do K = $L_K$, $U_K$
      do J = JJ, JJ+$B_{JJ}$-1
        do I = II, min(J, II+$B_{II}$-1)
          loop body
...
enddo
```

(b) After applying ISS

```
do JJ = $L_{JJ}$, $U_{JJ}$, $B_{JJ}$
  do II = $L_{II}$, min($U_{II}$, JJ-$B_{II}$+1), $B_{II}$
    do K = $L_K$, $U_K$
      do J = JJ, min(II+$B_{II}$-2, JJ+$B_{JJ}$-1)
        do I = II, J
          loop body
        enddo
      enddo
      do J = JJ, JJ+$B_{JJ}$-1              ┐
        do I = II, II+$B_{II}$-1            │partition A
          loop body                         ┘
    ...
  enddo
  do II = II, $U_{II}$, $B_{II}$
    do K = $L_K$, $U_K$
      do J = JJ, min(II+$B_{II}$-2, JJ+$B_{JJ}$-1)
        do I = II, J
          loop body
        enddo
      enddo
      do J = II+$B_{II}$-1, JJ+$B_{JJ}$-1
        do I = II, II+$B_{II}$-1
          loop body
    ...
  enddo
enddo
```

(c) After distributing loop K

```
do JJ = $L_{JJ}$, $U_{JJ}$, $B_{JJ}$
  do II = $L_{II}$, min($U_{II}$, JJ-$B_{II}$+1), $B_{II}$
    do K = $L_K$, $U_K$
      do J = JJ, min(II+$B_{II}$-2, JJ+$B_{JJ}$-1)
        do I = II, J
          loop body
        enddo
      enddo
    enddo
    do K = $L_K$, $U_K$                     ┐
      do J = JJ, JJ+$B_{JJ}$-1              │
        do I = II, II+$B_{II}$-1            │partition A
          loop body                         ┘
    ...
  enddo
  do II = II, $U_{II}$, $B_{II}$
    ...
  enddo
enddo
```

Figure 3.9: (a) Example of loop nest. (b) Loop nest after applying ISS repeatedly. (c) Loop nest after distributing loop K.

Now UCLs I and J in partition A can be fully unrolled. However, we cannot apply scalar replacement because they are not directly surrounded by a non-UCL loop. Thus, we have to distribute loop K (the innermost non-UCL loop) in the first loop nest to be able, later on, to apply scalar replacement in this partition. After distributing loop K we obtain the code shown in Fig. 3.9c (for the sake of clarity, we only show the first partition of loop II). Now, UCLs I and J in partition A can be fully unrolled and scalar replacement can also be applied.

Putting it All Together

In this section we will explain the complete algorithm for the Index Set Splitting process. The algorithm is shown in Fig. 3.10.

We call AN ("Active Nest") the loop nest where we want to fully unroll the UCLs, that is, the loop nest where ISS is applied. Initially the Active Nest is the loop nest after tiling the iteration space (variable OL in Fig. 3.10). We create the list of restrictions LL sorted according to the order described previously and we deal one by one with all of them. Each time we apply ISS to a loop I_p, according to a restriction R, we save the partition where the applied restriction does not hold in the list of loop nests pending to be processed, list LB in the code. When a restriction splits a UCL a new restriction appears. In this case, we insert the new restriction in the sorted list of restrictions (LL) 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 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 permutation transformation to make sure the UCLs become innermost loops before fully unrolling them. This loop permutation transformation can be directly performed because no inner *element* loop can have bound components that are affine functions of the UCLs.

Finally, if the loops that can be fully unrolled are not directly surrounded by another loop, loop distribution is applied to the innermost non-UCL loop.

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.

3.2.4 Complexity and Code Expansion

In this section, we give analytical expressions both for the complexity of our method and for the amount of code generated. We measure the complexity in number of times ISS has to be performed and the amount of code generated in number of loop nests (or partitions) 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. We note, however, that tiling in more than two dimensions for the register level can be counterproductive since the number of machine registers is usually small. Therefore, the expressions given assuming only two UCLs are likely to be the most common ones.

The number of times that our algorithm performs ISS depends on the number of bound components of the UCLs in the tiled code (just after the iteration space tiling phase). 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

```
Algorithm
INPUT:   OL   /* loop nest after tiling */
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};                    /* remove AN from LB */

       Create the sorted list LL of restrictions determined by the UCLs in AN;
       while (LL is not empty)
       {  R = first restriction of list LL;
          $I_p$ = loop to be split according to restriction R;
          Split loop $I_p$ of AN according to R;
          if  (R is an upper restriction)          /* R has the form $I_p \le u'$ */
          {  AN = 1st partition;
             LB = LB + {2nd partition};
          }
          else                        /* R is a lower restriction with the form $I_p \ge l'$ */
          {  AN = 2nd partition;
             LB = LB + {1st partition};
          }
          if (loop $I_p$ is an UCL)    Insert new restriction into sorted list LL;
          LL = LL - {R};          /* remove R from LL */
       }

       Permute loops of AN if necessary;              /* UCLs must be in the innermost positions */
       Distribute the innermost non-UCL loop if necessary;  /* UCLs must be directly surrounded
                                                               by a non-UCL loop */
}
endAlgorithm
```

Figure 3.10: Algorithm for the Index Set Splitting process.

outer UCL[4] and the S bound components are not. Neither R nor S+M contain the bound components that we want to be the *effective* ones.

As seen in the previous section, our algorithm applies first ISS to UCL loops. If the innermost UCL has M bound components that are affine functions of the outer UCL, we have M restrictions that split the outer UCL, and therefore, ISS has to be applied M times to this UCL. After dealing with these M restrictions, the iteration space is divided into M+1 partitions.

In only one of the new M+1 partitions the bound components of the inner UCL do not depend on the iteration variable of the outer UCL, therefore this is the only partition where the UCLs can be fully unrolled. In this particular partition, the outer UCL will have its R original bound components and M new ones that have appeared. The inner UCL will only have S bound components. Recall that every time index set splitting is applied to a loop, a bound component of a UCL disappears and a new component appears in the loop being split.

Nevertheless, the M new bound components that appear in the outer UCL might make other bound components become redundant. In the best case, the outer UCL will only have R bound components after dealing with the M restrictions and, in the worst case, it will have R+M bound components. We will develop the expressions for the worst case and, at the end of this section, we give the expressions for both the worst and the best cases.

After dealing with the M restrictions, we deal with the restrictions that do not split UCLs, from outermost to innermost. We apply R+M+S times ISS to achieve the partition where both UCLs can be fully unrolled and the number of loop nests obtained is (R+M+S+1). Each one of these loop nests has (M+1) loop nests inside (the partitions generated when dealing with the first M restrictions).

In only one of the (R+M+S+1) loop nests both UCLs can be fully unrolled. In the other loop nests, we will be able to unroll one (or none) of the two innermost loops by applying again ISS (repeatedly) to each of these loop nests. The total number of times that ISS is applied to each of these loop nests is (M+R)*S times. At each of the loop nests ISS is performed different number of times; it depends on the number of bound components that the UCL has. However, it can be demonstrated that the sum of the number of times ISS is performed on all these loop nests is (M+R)*S. The demonstration is shown in Appendix B.

Thus, at the end of the process, ISS has been performed M+(R+M+S)+((M+R)*S) times, in the worst case. The total number of loop nests generated is the number of loop nests generated when dealing with the M first restrictions ((M+1) loop nests) multiplied by the number of loop nests generated when dealing with restrictions that do not split UCLs ((R+M+S)+((M+R)*S)+1 loop nests), that is, (M+1)*((R+M+S)+((M+R)*S)+1).

4.From now on, we will say that a bound component depends on a particular loop iteration variable if it is an affine function of the loop iteration variable.

Times ISS has been performed (N_{iss})			worst case	M+R+M+S+(R+M)*S
			best case	M+R+S+R*S
Number of loop nests	**Total (N_{loop_nests})**			(M+1)*(N_{iss}-M+1)
	fully unrolled loops	2		1
		1	worst case	R+M+S
			best case	R+S
		0	worst case	$N^{worst}_{loop\_nests}$-1-(R+M+S)
			best case	$N^{best}_{loop\_nests}$-1-(R+S)

Table 3.2: Expressions to compute the maximum (worst case) and minimum (best case) number of times that ISS can be performed (rows 1 and 2), the total number of loop nests generated, as a function of N_{iss}, (row 3), the number of loop nests where the two innermost loops can be fully unrolled (row 4), the maximum and minimum number of loop nests where only one loop can be fully unrolled (rows 5 and 6) and the maximum and minimum number of loop nests where no loop can be fully unrolled (rows 7 and 8).

Table 3.2 shows the expressions to compute the maximum (worst case) and minimum (best case) number of times that our algorithm performs ISS (N_{iss}) to achieve that the UCLs in all partitions can be fully unrolled. Table 3.2 also shows the total number of loop nests generated (N_{loop_nests}) as a function of N_{iss} and the number of loop nests where two, one or no inner loop can be fully unrolled.

We conclude this section by indicating that our method increases code size and this fact could increase the instruction cache misses and potentially decrease performance. Nevertheless, we will see later in Section 3.5 that the generated code has a good degree of locality and that the overall instruction cache misses are insignificant. In Section 3.5 we will also present some experimental data on the number of times ISS has to be applied and the number of loop nests generated using as benchmarks typical linear algebra programs.

3.3 BENEFITS OF REGISTER TILING

Tiling is a loop transformation that has been mostly used to exploit data reuse at the different memory levels. The desired effect of applying tiling to a certain memory level is to reduce the number of requests issued to the next level in the hierarchy. For example, when applying tiling to a certain cache level, we are reducing the number of misses in that level, i.e., reducing the number of requests that this level makes to the following level in the hierarchy.

When applying tiling to the register level, what we achieve is to reduce the number of requests to the first cache level, that is, to reduce the absolute number of loads/store instructions performed by a program. Moreover, tiling for the register level has an extra advantage that can not be achieved by tiling for the cache levels: it improves the intra-iteration ILP. In this section we show how tiling for the register level can expose both advantages.

(a) Original code

```
do I = 1, N
  do J = 1, N
    RR1 = C(I,J)
    do K = 1, N
      RR1 =RR1 + A(I,K) *D(K,J)
    enddo
    C(I,J) = RR1
  enddo
enddo
```

(b) Tiled code

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do K = 1, N
      RR1 = RR1 + A(II,K) * D(K,JJ)
      RR2 = RR2 + A(II,K) * D(K,JJ+1)
      RR3 = RR3 + A(II+1,K) * D(K,JJ)
      RR4 = RR4 + A(II+1,K) * D(K,JJ+1)
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
  enddo
enddo
```

Figure 3.11: Code of the matrix product (a) after applying scalar replacement, (b) after tiling for the register level at two dimensions of the iteration space.

3.3.1 Reducing Load/Store Instructions

Data reuse at the register level essentially translates into a reduction of the number of load/store instructions. Reducing the number of load/store instructions reduces the pressure on the fetch unit, reduces data memory traffic and, indirectly, might also decrease data cache misses. To illustrate this reduction of load/store instructions we will use as an example a square matrix product code. Figure 3.11a and 3.11b present the original code after applying scalar replacement and the code after tiling for the register level at two dimensions of the iteration space, respectively. For simplicity, we assume that N is multiple of B_{II} and B_{JJ} and $B_{II}=B_{JJ}=2$.

In the original code (Fig. 3.11a) 2 loads are performed in each iteration of the loop body. One iteration of the loop body in the tiled code (Fig. 3.11b) performs 4 iterations of the original loop body and, yet, only 4 loads are executed. This represents an average of only 1 load per iteration of the original loop body. Note that tiling for the register level also exploits data reuse inside the loop body. In the example, it exploits the reuse of A(II,K), A(II+1,K), D(K,JJ) and D(K,JJ+1). The data reuse exploited across iterations of the inner loop (matrix C) is achieved by the scalar replacement transformation and we note that both codes (original and tiled) achieve the same degree of reuse of matrix C.

An important consequence of reducing the number of loads and stores instructions is an improvement in loop balance. The relation between memory accesses and floating-point operations inside the loop body determines the machine resource that is the bottleneck when the loop is executed. High performance can be achieved in a particular machine if it can operate in a steady manner with both memory accesses and floating-points operations being performed at peak speed. To quantify this relationship, Callahan et. al. [21] define the notion of *machine balance* (β_M) as the rate at which data can be fetched from memory (M_M), compared to the rate at which floating-point operations can be performed (F_M):

$$\beta_M = \frac{\text{max words/cycle}}{\text{max flops/cycle}} = \frac{M_M}{F_M}$$

The values of M_M and F_M represent peak performance, where the size of a word is the same as the precision of the floating-point operations.

Just as machines have balance ratios, so do loops. The balance ratio of a specific loop is defined as follows:

$$\beta_L = \frac{\text{number of memory references}}{\text{number of flops}} = \frac{M_L}{F_L}$$

Comparing β_M to β_L can give us a measure of the performance of a loop running on a particular architecture. If $\beta_M = \beta_L$, the loop is *balanced* for the machine and will run well on that particular machine. If $\beta_M < \beta_L$, then the loop needs data at a higher rate than memory system can provide and idle computational cycles will exist. Such a loop is said to be *memory bound* and its performance can be improved by lowering β_L. If $\beta_M > \beta_L$, then data cannot be processed as fast as it is supplied to the processor and memory bandwidth will be wasted. Such a loop is said to be *compute bound* and its performance cannot be improved since floating-points operations usually cannot be removed.

To achieve high performance on a particular machine it is convenient to have *balanced* or *compute bound* loops. Tiling for the register level is a transformation that can improve the performance of memory-bound loops by converting them into balanced or compute-bound loops. Tiling for the register level introduces more computation into the loop body without a proportional increase in memory references, and therefore, it has the potential to improve loop balance.

As an example, consider the original code of the matrix product of Fig. 3.11a, after applying scalar replacement. In this code, we have two floating-point operations and two memory references, giving a loop balance of 1. On a machine that can perform twice as many floating-point operations as memory accesses per clock cycle, such as the MIPS R10000, this loop is memory-bound and does not run at peak machine speed. However, after tiling for the register level (see Fig. 3.11b), we obtain a new loop body that contains 8 floating-point operations and only 4 memory accesses, giving a loop balance of 0.5. Now, the transformed loop is balanced for an R10000 processor and would perform better than the original code.

Having balanced or compute-bound loops is not enough for running at peak speed. It is also necessary to have enough parallelism at the instruction level, so that the processor can issue memory accesses and floating-point instructions in parallel. In the next section, we show how tiling for the register level also exposes ILP.

3.3.2 Improving Instruction Level Parallelism

In single loops, compilers can extract ILP by performing software pipelining and/or unrolling the innermost loop. Both techniques draw out parallelism between iterations of the original loop body, but they are implemented in different ways. Inner unrolling replaces the body of the loop by several copies of the body and adjusts the loop-control code accordingly. Software pipelining, instead, reorganizes loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. Nonetheless, these two transformations are limited by the recurrences[5] of the loop body.

In nested loops, however, there are more opportunities to expose ILP. In particular, multi-dimensional tiling for the register level always exposes ILP, regardless if the unrolled loops add new dependences or not in the new loop body. We illustrate this fact with the following example.

Suppose we have a 3-deep loop nest that has been tiled in two dimensions with a tile size of 4x4 (Fig. 3.12). The code after tiling and fully unrolling the loops contains in its body 16 instances of the original loop body. Three different situations can happen: 1) None of the unrolled loops adds a dependence. In this case, the 16 instances in the unrolled loop body can be performed in parallel; 2) One of the unrolled loops adds dependences. In this case every 4 instances can be performed in parallel; 3) Both unrolled loops add dependences. In this case we have a non-uniform parallelism, but some of the instances can be performed in parallel.

Figure 3.12 shows the unrolled loop body and, for each of the three previous cases, we show the dependences and mark with a filled rectangle the original iterations that can be performed in parallel. Each point represents one iteration of the original loop body. For simplicity we consider all operations

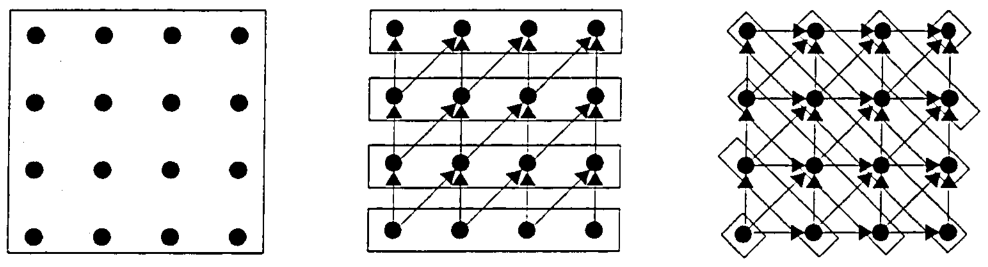

(a) (b) (c)

Figure 3.12: ILP obtained when (a) none of the fully unrolled loops carried dependences, (b) one of the unrolled loops carried dependences and (c) both of the unrolled loops carried dependences.

5.There is a recurrence in the loop body if the innermost loop carries a dependence.

performed in one iteration of the original loop body as a unit. However, the instruction scheduler can also extract ILP between multiple operations of one iteration. Finally, recall that we are assuming fully permutable loop nests (if not, loop tiling is not a legal transformation). Therefore, all components of all dependence vectors are greater or equal to 0 (see Chapter 2, Section 2.3.1).

Thus, tiling more than one dimension for the register level, regardless of the loops being tiled, always achieves a certain amount of ILP. Moreover, the cases where maximum ILP is not achieved are caused by true, anti or output dependences which intrinsically limit ILP. However, having more dependences indicates that we have more data reuse and therefore, the loss of ILP in these cases can be compensated by the increase in data reuse. We will see later in Section 3.5 that the amount of ILP achieved is sufficient for obtaining high performance in modern processors.

3.3.3 Register Tiling vs. Outer Unrolling

There is another loop transformation which is able both to improve ILP and to enhance data locality, namely *outer unrolling* (also called *unroll-and-jam*) [20][23][26]. It consists in unrolling an outer loop in a nest and then jamming the resulting inner loops back together. Thus, outer unrolling enhances data locality at the register level in the unrolled dimension of the iteration space. Actually, applying outer unrolling is equivalent to tiling only one dimension of the iteration space.

How does tiling for the register level compare to outer unrolling? Outer unrolling can also exploit data reuse inside the loop body, but, since it only enhances locality in the unrolled dimension, it needs more registers to achieve the same reuse as tiling more than one dimension. Figure 3.13a shows the code of the matrix product previously presented in Fig. 3.11, after applying outer unrolling

(a) Outer unrolling

```
do II = 1, N, B_II
  do J = 1, N
    RR1 = C(II,J)
    RR2 = C(II+1,J)
    RR3 = C(II+2,J)
    RR4 = C(II+3,J)
    do K = 1, N
      RR1 = RR1 + A(II,K) * D(K,J)
      RR2 = RR2 + A(II+1,K) * D(K,J)
      RR3 = RR3 + A(II+2,K) * D(K,J)
      RR4 = RR4 + A(II+3,K) * D(K,J)
    enddo
    C(II,J) = RR1
    C(II+1,J) = RR2
    C(II+2,J) = RR3
    C(II+3,J) = RR4
  enddo
enddo
```

(b) Register Tiling

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do K = 1, N
      RR1 = RR1 + A(II,K) * D(K,JJ)
      RR2 = RR2 + A(II,K) * D(K,JJ+1)
      RR3 = RR3 + A(II+1,K) * D(K,JJ)
      RR4 = RR4 + A(II+1,K) * D(K,JJ+1)
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
  enddo
enddo
```

Figure 3.13: Code of the matrix product (a) after applying outer unrolling with an unroll factor of 4 (B_{II}=4) and (b) after tiling for the register level at two dimensions of the iteration space.

to loop I with an unrolling factor of 4 (B_{II}=4) (for easing the comparison, we also include again the tiled code). In this code (Fig. 3.13a), 1.25 loads per iteration of the original loop body are performed, achieving less data reuse than the tiled code of Fig. 3.11b. Moreover, outer unrolling needs 9 registers (1 for D(K,J), 4 for A(II:II+3,K) and 4 for C(II:II+3,J)) while the tiled code only needs 8 registers (2 for A(II:II+1,K), 2 for D(K,JJ:JJ+1) and 4 for C(II:II+1,JJ:JJ+1)).

The important point to note is that using less registers, multi-dimensional tiling achieves more data reuse than outer unrolling, that is, multi-dimensional tiling achieves more data reuse that tiling only one dimension of the iteration space. This is especially important at the register level, since being able to fully exploit the (small) storage space available at this level can mean a big performance difference. Moreover, multi-dimensional tiling reduces the register pressure and, therefore, the use of aggressive scheduling techniques for exploiting instruction level parallelism is not overly limited.

Finally, we want to note that outer unrolling can be applied repeatedly to several loops in a nest and, in this case, data reuse is exploited in all the unrolled dimensions. Thus, applying unroll-and-jam to several loops in a nest is comparable to multi-dimensional tiling. However, to the best of our knowledge, there is no previous work proposing a technique to apply outer unrolling to several loops in arbitrary, non-rectangular, loop nests. Later, in Section 3.6, we will discuss this claim more deeply.

3.4 LOCALITY ANALYSIS FOR THE REGISTER LEVEL

One challenge that confronts a compiler that tries to apply loop tiling is determining which loops are the best ones to be tiled, the tile sizes in each dimension and the order of the loops that deliver the best performance. Previous work on determining the tiling parameters has been mostly focused at the cache level [31][80][83][110][121], and less attention has been paid to the register level. Although the basic principles in memory hierarchy optimization are similar for all levels, each level has slightly different characteristics, requiring slightly different considerations. For example, registers typically cannot take advantage of spatial reuse[6], so we only have to consider the temporal locality of the references and we must take into account both read and write references when optimizing for them.

In this section we discuss how the locality analysis for the register level has to be carried out and propose a simple heuristic to decide the tiling parameters for this level. Note that the algorithm proposed in previous sections to perform register tiling is a high-level (source to source) transformation. Of course, working at the source level prevents us from controlling many of the low level transformations typically performed by the compiler's back-end (instruction scheduling, register allocation, etc.). Therefore, the heuristic presented has been geared towards simplicity rather than trying to find optimal tiling parameters, since there are too many aspects of the code generation process that escape from our control.

6.The intel i860 can load two registers with adjacent memory locations, which allows for some exploitation of spatial locality.

3.4.1 Tile Directions

Exploiting data reuse in more than one dimension of the iteration space, whenever possible, improves the performance of the memory hierarchy [122]. Previous work on tiling for the cache level [122][128] has almost always assumed that, if in a n-dimensional iteration space all loops carry reuse, then all loops should be tiled and the tile size has to be chosen so that all data being referenced inside the tile fits in the memory level being exploited. This way of selecting the tile sizes results in all possible orders of the element loops yielding the same degree of locality. However, there are other works [98][97] that state that if in a n-dimensional iteration space all loops carry reuse, then *n-1* loops should be tiled. Not tiling one loop that carries data reuse and establishing a proper order of the inner loops yields bigger tile sizes and, therefore, more data locality, than tiling all loops that carry reuse. This fact is especially important at the register level, since being able to fully exploit all the (small) storage space available at this level can make a big performance difference.

Let's now see how bigger tile sizes can be achieved (and thus, more data reuse) by taking into account the ordering of the element loops and by not tiling all directions that carry reuse. We will use an example to illustrate this point. Figure 3.14a shows the code of the square matrices product, where all three loops carry self-temporal reuse. Since registers typically cannot take advantage of spatial reuse, we only need consider temporal reuse when optimizing for them. In direction I matrix D is reused, in direction J matrix A is reused and in direction K matrix C is reused. After tiling the iteration space at all three dimensions we obtain the code of Fig. 3.14b. The tile size is $B_{II} \times B_{JJ} \times B_{KK}$ and, for simplicity, we assume N is multiple of B_{II}, B_{JJ} and B_{KK}. For properly exploiting the register level, it is necessary to fully unroll the element loops and to apply scalar replacement. The resulting code, assuming $B_{II}=B_{JJ}=B_{KK}=2$, is shown in Fig. 3.14c.

The tiled code of Fig. 3.14b works well for any order of the element loops since the following submatrices fit into the memory level being exploited:

$$C(II:II+B_{II}-1,JJ:JJ+B_{JJ}-1),\ A(II:II+B_{II}-1,KK:KK+B_{KK}-1),\ D(KK:KK+B_{KK}-1,JJ:JJ+B_{JJ}-1)$$

In particular, at the register level and with $B_{II}=B_{JJ}=B_{KK}=2$, we need 12 register to keep all the data that is referenced inside the tile. As shown in Fig. 3.14c, we need 4 registers for RR1, RR2, RR3, RR4, and 8 more registers to keep the data belonging to matrices A and D that are referenced in the loop body.

The order of the tile loops or, more exactly, the most internal tile loop, determines which of the previous data blocks are reused in the next tile. In Fig. 3.14c, we select loop KK to be the innermost tile loop for achieving maximum data reuse. The registers holding RR1, RR2, RR3 and RR4 (matrix C) are reused in every iteration of loop KK. Thus, they are only loaded and stored (N/2) x (N/2) times. The other 8 registers are only reused inside each iteration of loop KK and, therefore, they are loaded in each iteration. The total number of loads and stores of code of Fig. 3.14c is N^3+N^2 loads and N^2 stores.

(a) Original code

```
do I = 1, N
  do J = 1, N
    do K = 1, N
      C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

(b) Three-dimensional iteration space tiling

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    do KK = 1, N, B_KK
      do I = II, II+B_II-1
        do J = JJ, JJ+B_JJ-1
          do K = KK, KK+B_KK-1
            C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

(c) Register tiling

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do KK = 1, N, B_KK
      RR1 = RR1 + A(II,KK) * D(KK,JJ)
      RR1 = RR1 + A(II,KK+1) * D(KK+1,JJ)
      RR2 = RR2 + A(II,KK) * D(KK,JJ+1)
      RR2 = RR2 + A(II,KK+1) * D(KK+1,JJ+1)
      RR3 = RR3 + A(II+1,KK) * D(KK,JJ)
      RR3 = RR3 + A(II+1,KK+1) * D(KK+1,JJ)
      RR4 = RR4 + A(II+1,KK) * D(KK,JJ+1)
      RR4 = RR4 + A(II+1,KK+1) * D(KK+1,JJ+1)
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
  enddo
enddo
```

Figure 3.14: (a) Form IJK of the square matrix product. (b) Code after tiling the iteration space at all three dimensions. (c) Code after fully unrolling and applying scalar replacement (register tiling).

Now we will show that, if we fix the order of the element loops so that the outermost loop is the one corresponding to the innermost tile loop, we can do the tile size bigger [97]. In the code of Fig. 3.14b, we can select the order KIJ for the element loops. Using this order, it is not necessary to keep all data referenced inside the tile into the memory level being exploited. It is sufficient to keep C(II : II+B_{II}-1,JJ : JJ+B_{JJ}-1), A(II : II+B_{II}-1,KK) and D(KK,JJ : JJ+B_{JJ}-1), that is, a submatrix of C, a subrow of D and a subcolumn of A. Each subcolumn of matrix A referenced inside the tile can share the same registers (or even the same memory locations if applying this idea to a cache level) and each row of D can also share the same registers (or memory locations). Therefore, in our example, we would only need 8 registers. The 8 register are: Four for C(II,JJ), C(II,JJ+1), C(II+1,JJ), C(II+1,JJ+1), two for D(KK,JJ) and D(KK,JJ+1) (D(KK+1,JJ) and D(KK+1,JJ+1) would share the same registers) and two for A(II,KK) and A(II+1,KK) (A(II,KK+1) and A(II+1,KK+1) would share the same registers). Figure 3.15a shows the tiled code of the square matrix product with the order of the element loops fixed to KIJ, and without unrolling loop K. The 4 different rectangles indicate the data references that share the same registers inside the loop body. Note that even A(II,KK) and A(II+1,KK) could share the same register, reducing even more the number of registers used.

(a) Register tiling without unrolling loop K

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do KK = 1, N, B_KK
      do K = KK, KK+B_KK-1
        RR1 = RR1 + A(II,K) * D(K,JJ)
        RR2 = RR2 + A(II,K) * D(K,JJ+1)
        RR3 = RR3 + A(II+1,K) * D(K,JJ)
        RR4 = RR4 + A(II+1,K) * D(K,JJ+1)
      enddo
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
  enddo
enddo
```

(b) Code after coalescing loops KK and K

```
do II = 1, N, B_II
  do JJ = 1, N, B_JJ
    RR1 = C(II,JJ)
    RR2 = C(II,JJ+1)
    RR3 = C(II+1,JJ)
    RR4 = C(II+1,JJ+1)
    do K = 1, N
      RR1 = RR1 + A(II,K) * D(K,JJ)
      RR2 = RR2 + A(II,K) * D(K,JJ+1)
      RR3 = RR3 + A(II+1,K) * D(K,JJ)
      RR4 = RR4 + A(II+1,K) * D(K,JJ+1)
    enddo
    C(II,JJ) = RR1
    C(II,JJ+1) = RR2
    C(II+1,JJ) = RR3
    C(II+1,JJ+1) = RR4
  enddo
enddo
```

Figure 3.15: (a) Code of the square matrix product after tiling for the register, fixing the order KIJ of the element loops. (b) Code after coalescing loops KK and K.

Note now that loop KK (the innermost tile loop) and loop K (the outermost element loop) are adjacent and therefore they can be coalesced; thus, it was not necessary to tile loop K in the original code. Figure 3.15b shows the code if we do not tile loop K in the original code. Note that in this code we only need 8 registers to keep all data that is reused and four of the 8 registers are reused in every iteration of loop K (the non-tiled loop). In this case, the total number of loads and stores is again N^3+N^2 loads and N^2 stores. The important point to note is that, using less registers, we have achieved the same degree of reuse as the code in Fig. 3.14c. Therefore, not tiling a loop that carries reuse reduces register pressure while maintaining the same degree of locality than tiling all loops. Moreover, since we need less data in registers, we could do the tile size bigger if there are registers in excess, and thus, we could increase data locality. For example, if we chose a tile size such as $B_{II}=2$ and $B_{JJ}=3$, we would need 11 registers and the total number of loads and stores would be $(5/6)N^3+N^2$ loads and N^2 stores.

Now, the locality analysis for the register level consists on determining which loop should not be tiled. Once we decide which loop should not be tiled, the order of the loops must be the following: the *tile* loops are the outermost loops, the *non-tiled* loop is the next loop and the *element* loops are the innermost ones (see Fig. 3.16). Note that, at the register level, once the non-tiled loop is chosen, the relative order of the remaining tile and element loops is indifferent. The order of the tile loops is not important, because there is no reuse between iterations of the tile loops. The order of the element loops is no longer relevant, since they will be later fully unrolled.

```
do II = 1, N, B_II                      ] Tile loops
  do JJ = 1, N, B_JJ                    ]
    do K = 1, N                         ] Non-tiled loop
      do I = II, II+B_II-1              ] Element loops
        do J = JJ, JJ+B_JJ-1            ]
          C(I,J) = C(I,J) + A(I,K) * D(K,J)
...
enddo
```

Figure 3.16: Example of loop nest.

3.4.2 Iteration Space Shape

At first sight, register tiling should be performed so that whichever loop carries the most reuse is not tiled. This way, register reuse is maximized and the number of load/store instructions executed is minimized. However, if we only consider reuse directions and do not take into account the iteration space shape, the tiled loop nest can suffer performance degradation due to the time wasted executing boundary-tiles. In this section, we will show how the iteration space shape can affect processor performance.

As shown in Section 3.2, after applying 2-dimensional register tiling to a 3-deep loop nest, we obtain a new code that contains: one loop nest where the two element loops are fully unrolled (we will refer to it as the *fully unrolled loop nest*), some loop nests where only one of the element loops is fully unrolled (we will refer to them as *partially unrolled loop nests*) and some loop nests where no loop is fully unrolled (*non-unrolled loop nest*). The fully unrolled loop nest only traverses core-tiles, while the other nests traverse boundary-tiles. Obviously, boundary-tiles cannot be executed at the same speed as core-tiles, because in boundary-tiles less ILP and less data reuse is achieved. Moreover, boundary-tiles traversed by *partially unrolled loop nests* are executed at higher speed than boundary-tiles traversed by *non-unrolled loop nest*. Thus, to reduce the execution time of boundary-tiles it is preferable to traverse them using *partially unrolled loop nests*.

How the boundary-tiles are traversed after register tiling depends on the non-tiled loop selected in the locality analysis. As an example, consider the code of the SSYRK routine from BLAS, shown in Fig. 3.17a, that describes the non-rectangular iteration space shown in Fig. 3.17b. In the SSYRK routine, the loop that provides more temporal reuse is loop K. If we make loop K innermost and apply scalar replacement, the loop body only executes two loads per iteration, while making loops I or J innermost will generate a loop body performing two loads and one store per iteration.

(a) SSYRK routine

```
do J = 1, N
  do K = 1, N
    do I = J, N
      C(I,J) = C(I,J) + A(J,K) * A(I,K)
  . . .
enddo
```

(b) Iteration space

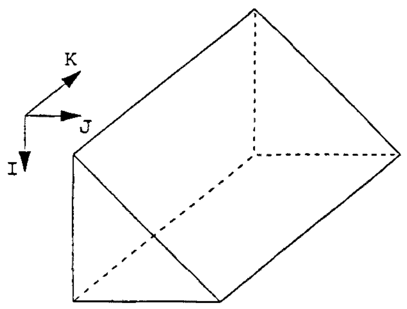
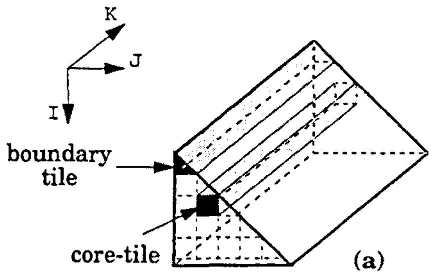


Figure 3.17: (a) Code of the SSYRK routine from BLAS. (b) Iteration space described by the SSYRK routine.

Let's now see what happens if we select loop K (the loop carrying most reuse) to be the non-tiled loop. After register tiling, the iteration space is divided as shown in Fig. 3.18a. Figure 3.18b shows the code after the iteration space tiling phase. Directions I and J are tiled and the tiles are executed along direction K. Each square tile is a core-tile that is fully unrolled in both directions I and J. The remainder of the tiles are boundary-tiles that cannot be unrolled in both directions. Two different types of boundary-tiles have been generated: One of them (type 1 in Fig. 3.18c) has been generated because N might be a non-multiple of the tile size in the I dimension and the other one (type 2) has been generated because, in the tiled code, there is a bound of loop I that depends on loop J that makes the space defined by these two loops to be non-rectangular. The boundary-tiles of type 1 are traversed by a *partially unrolled loop nest.* Note that in those tiles, loop J always executes as many iterations as the tile size in its dimension, and therefore, it can be fully unrolled. However, all boundary-tiles of



(b) Tiled code

```
do JJ = 1, N, $B_{JJ}$
  do II = max(1, JJ), N, $B_{II}$
    do K = 1, N
      do J = max(1, JJ), min(N, JJ+$B_{JJ}$-1, II+$B_{II}$-1)
        do I = max(J, II), min(N, II+$B_{II}$-1)
          loop body
      . . .
enddo
```

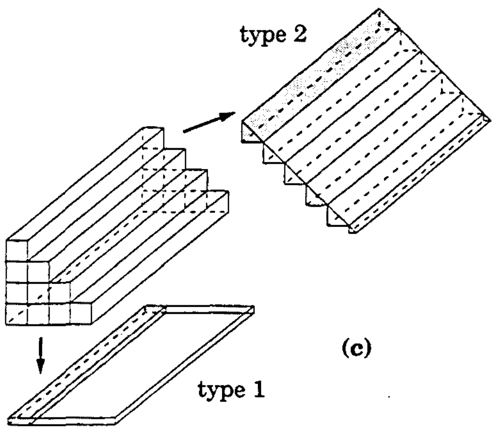
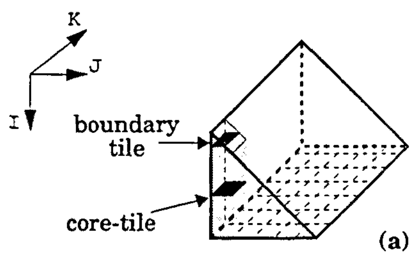


Figure 3.18: (a) Iteration space of the SSYRK routine after applying register tiling. Loop K is the non-tiled loop and directions I and J are tiled. (b) Code after the iteration space tiling phase. (c) Boundary-tiles of the tiled iteration space.



(b) Tiled code

```
do KK = 1, N, $B_{KK}$
  do JJ = 1, N, $B_{JJ}$
    do I = max(1, JJ), N
      do K = max(1, KK), min(N, KK+$B_{KK}$-1)
        do J = max(1, JJ), min(I, JJ+$B_{JJ}$-1)
          loop body
  ...
enddo
```

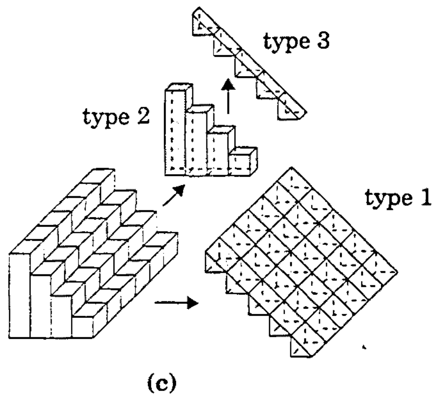


Figure 3.19: (a) Iteration space of the SSYRK routine after applying register tiling. Loop I is the non-tiled loop and directions K and J are tiled. (b) Code after the iteration space tiling phase. (c) Boundary-tiles of the tiled iteration space.

type 2 are traversed by *non-unrolled loop nests*, since neither J nor I execute as many iterations as the tile size in their dimensions. All these boundary-tiles of type 2 are executed without an increase in ILP and data reuse (with respect to the original code), therefore performance might not be improved if the number of boundary-tiles traversed by *non-unrolled loop nests* is large.

Let's now see what happens if we select loop I to be the non-tiled loop. After register tiling, the iteration space is divided as shown in Fig. 3.19a. Figure 3.19b shows the code after the iteration space tiling phase. Directions K and J are tiled and the tiles are executed along direction I. In this case, there are three different types of boundary-tiles: two of them (types 2 and 3 in Fig. 3.19c) are generated because N might be a non-multiple of the tile sizes and the other one (type 1) is generated because, in the tiled code, there is a bound of loop J that depends on the non-tiled loop I that makes the space defined by these two loops to be non-rectangular. In this example, the boundary-tiles of types 1 and 2 are traversed by *partially unrolled loop nests*. In those tiles, either loop J (type 2) or loop K (type 1) always execute as many iterations as the tile size in their dimension, and therefore, they can be fully unrolled. However, the boundary-tiles of type 3 are traversed by a *non-unrolled loop nest*, since neither J nor K execute as many iterations as the tile size in their dimensions. In this case, only these boundary-tiles of type 3 are executed without an increase in ILP and data reuse with respect to the original code.

Comparing both examples of Fig. 3.18 and 3.19, it can be seen that depending on the non-tiled loop selected in the locality analysis, the number of boundary-tiles traversed by non-unrolled loop nests varies significantly. How does this fact affect processor performance? Figure 3.20 shows the performance obtained in the SSYRK program using different criterions to select the non-tiled loop at the register level. TRL-I selects loop I as the non-tiled loop, so that the number of boundary-tiles executed by *non-unrolled loop nests* is minimized, while TRL-K selects loop K, so that maximum register reuse in the core-tiles is achieved. The tile sizes in both cases are 4x4 and the problem size is varied from 10 to 100. We use small problem sizes to avoid that higher memory levels affect processor performance. Measures were taken on two different processors: an ALPHA 21164 and a MIPS R10000 processor.

It can be seen that for almost all problem sizes, not tiling loop I yields better performance. Only for medium problem sizes (between 70 and 100) and only for the MIPS processor, TRL-I yields less performance than TRL-K, but the performance difference is very small. We note in passing that this loss in performance for this particular problem sizes is mostly due to the selected tile sizes. As we will see later, TRL-I can be still improved by choosing tile sizes according to the reuse carried by the tiled loops.

We can also see in Fig. 3.20 that the difference in performance between the two versions is greater for very small problem sizes than for medium problem sizes. For problem sizes that are very small and/or not multiple of the tile sizes, the execution time wasted on boundary-tiles is a significant fraction of the total execution time. If the boundary-tiles are mostly traversed by *partially unrolled loop nests* (like TRL-I version) instead of using *non-unrolled loop nests* (like TRL-K version), better performance is achieved. However, as the problem size increases, the execution time of boundary-tiles becomes less significant.

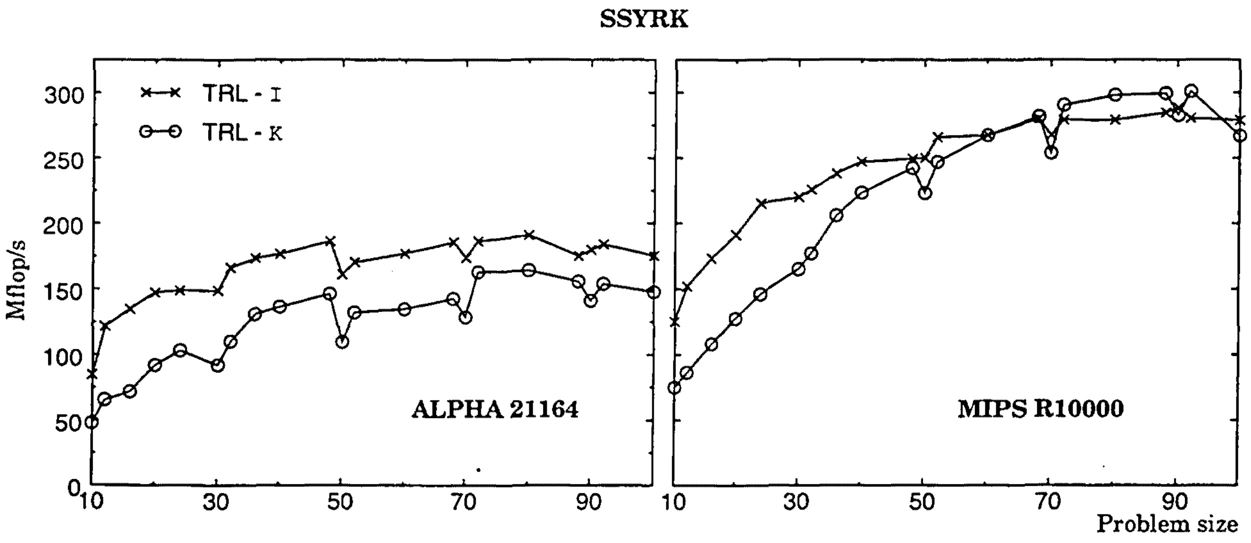


Figure 3.20: Performance of SSYRK on the ALPHA 21164 and the MIPS R10000 processor, varying the problem size from 10 to 100. TRL-I corresponds to register tiling selecting loop I as the non-tiled loop and TRL-K corresponds to register tiling selecting loop K as the non-tiled loop. The tile sizes in both cases are fixed to 4x4.

Moreover, the performance difference is more noticeable on the ALPHA processor than 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 processor is able to unroll the *element* loops of boundary-tiles dynamically, extracting more ILP from these tiles, and thus, executing the boundary-tiles faster.

Summarizing, we can conclude that for small problem sizes and/or for very irregular iteration space shapes, it is more important to execute boundary-tiles as fast as possible than to increase register reuse in the core-tiles. In the next section we propose a simple heuristic that considers the iteration space shape to decide which loop should be the non-tiled loop at the register level.

3.4.3 Heuristic

We have seen in Section 3.3 that register tiling provides two different advantages (register reuse and ILP) that vary depending on the tiling parameters. Finding the optimal tiling parameters for a given loop nest is a very complex and time-consuming problem that depends upon machine and program characteristics. Moreover, our proposal of register tiling is a source to source optimizing transformation that helps compilers generate efficient machine code. It does not control other low level compiler's optimizations such as instruction scheduling and register allocation, making it more difficult (if not impossible) to find the optimal tiling parameters. In this section, we develop a very simple heuristic to select the tiling parameters at the register level that behaves well for typical linear algebra problems.

To simplify the analysis we make the following assumptions:

First, we only consider n-dimensional iteration spaces with all loops carrying temporal data reuse. As mentioned in Chapter 2, this thesis only deals with numerical codes for which loop tiling has had the greatest success. Numerical codes, and specially linear algebra algorithms, usually consist of a n-deep loop nest while using $(n\text{-}1)$-dimensional data structures, yielding data reuse in several (typically all) dimensions of the iteration space. Anyway, if there were loops not carrying temporal reuse, it would be necessary to discard them (making them outermost) and apply register tiling to the other loops providing reuse.

Second, we always tile $(n\text{-}1)$ dimensions of the n-dimensional iteration space. As shown previously, not tiling one loop that carries data reuse reduces register pressure while maintaining data locality. Moreover, we have also seen that by tiling more than one dimension of the iteration space, regardless of the dimensions being tiled, a reasonable amount of ILP is achieved. However, we note that tiling more than two dimensions for the register level can be self-defeating in modern microprocessors, since the number of machine registers is usually small (typically 32), and tiled loop nests that spill floating-point registers excessively may suffer performance degradation. Thus, for loop nests deeper than 3, it would be necessary to discard the loops carrying less temporal reuse (making them outermost) and apply register tiling to the three loops providing most reuse.

Finally, we also assume that by tiling (*n-1*) dimensions, regardless of the dimensions being tiled, we obtain a balanced or compute-bound loop nest, so that the transformed loop nest can be run at peak speed. Since all loops carry data reuse, it is reasonable to assume that the fully unrolled loop nest generated after register tiling will be balanced or compute-bound. Thus, in our heuristic, we will not explicitly consider the amount of ILP and the loop balance that can be achieved. Instead, we focus on selecting the non-tiled loop that generates less boundary-tiles traversed by non-unrolled loop nests, so that the overhead of executing boundary-tiles is minimized. Only if there are several alternatives, we will select the loop that provides more temporal data locality, so that register reuse is improved.

Our heuristic to determine the tiling parameters is the following one:

1. Determine all different self and group temporal reuse vectors as proposed by [121] (explained in Chapter 2), discarding all reuse vectors that are not parallel to the iteration space axes.
2. Give each reuse vector a weight. This weight is the number of references in the original loop body that have generated this reuse vector, considering reads and writes. For group temporal reuse vectors, only one of each pair of references is counted.
3. For each loop in the nest being the non-tiled loop, determine the amount of boundary-tiles traversed by non-unrolled loop nests. Then, pick the loops that generate less boundary-tiles traversed by non-unrolled loop nests.
4. From all loops picked in step 3, select as non-tiled loop, the loop that carries most temporal reuse. This is determined by the weight of its corresponding reuse vector.
5. Compute the tile sizes in proportion to the quantity of reuse carried by each tiled direction, taking into account the number of available machine registers.

We only take into account reuse directions that are parallel to the iteration space axes, because the strip-mining and loop permutation transformations used to divide the iteration space into regular tiles, generate tiles whose boundaries are always parallel to the iteration space axes. For a 3-dimensional iteration space, for example, we can only exploit data reuse in directions (1,0,0), (0,1,0) or (0,0,1). Anyway, if the reuse vector with greater weight happened to be not parallel to the iteration space axes, we could apply a unimodular or non unimodular transformation [84] before tiling to make this direction become parallel to the iteration space axes.

Reuse vector (J,K,I)	**Weight**	**References**
(0,1,0)	2	C(I,J) (read and write)
(1,0,0)	1	A(I,K) (read)
(0,0,1)	1	A(K,J) (read)

Table 3.3: Reuse vectors, weight and references involved in the SSYRK routine.

Our heuristic computes first the temporal data reuse carried by each loop in the nest, giving to each reuse vector a weight. This weight is telling us how many load/store instructions will be saved if each loop is made the innermost loop in the nest. As an example of how to compute the weight, consider the SSYRK example used before (see Fig. 3.17). There are four memory references, namely C(I,J) (read), C(I,J) (write), A(I,K) and A(J,K). The self-temporal reuse vectors are (0,1,0), (1,0,0) and (0,0,1) and there is no group-temporal reuse vector (we note that the loop ordering is (J, K, I)). Table 3.3 shows the weight of each reuse vector and the references that have generated them. The reuse vector (0,1,0) has a weight of 2, since there are two references that have this reuse vector. Thus, the loop that carries most temporal reuse is loop K.

After computing the reuse carried by each loop in the nest, our heuristic determines the amount of boundary-tiles traversed by *non-unrolled loop nests* for each loop in the nest being the non-tiled loop. We will only count the number of (*n-1*)-dimensional hyperplanes where the boundary tiles cannot be unrolled. For a 3-dimensional iteration space, for example, we only count the number of (2-dimensional) planes. We will not consider the number of (1-dimensional) edges, because the time consumed executing edges is very small with respect to the total execution time. However, the execution time wasted on non-unrolled planes is a significant portion of the total execution time and they are the cause of performance degradation.

Given a loop nest and a particular non-tiled loop, the number of (*n-1*)-dimensional hyperplanes (containing non-unrolled boundary-tiles) that will be generated after tiling can be easily computed by projecting the iteration space along the non-tiled direction and counting the number of bounds of the inner loops that are affine function of outer loops.

As an example consider the SSYRK routine of Fig. 3.17a (page 99). Figure 3.21 shows, for each loop in the nest, the projection of the iteration space along its direction. The projection along direction I (and also along direction J) defines a rectangular space, and thus, there is no bound of the innermost loop that is affine function of the outermost loop. If we tile the projected iteration space at two dimensions, all tiles generated can be unrolled in one or both directions, except the tile at the bottom-right corner (marked in grey in Fig. 3.21). This means that there will be an edge in the tiled code traversed by a *non-unrolled loop nest*. However, the projection along direction K defines a non-rectangular space, because a lower bound of the innermost loop I is affine function of the

(a) Direction I

```
do K = 1, N
  do J = 1, N
  ...
enddo
```

(b) Direction J

```
do K= 1, N
  do I = 1, N
  ...
enddo
```

(c) Direction K

```
do J = 1, N
  do I = J, N
  ...
enddo
```

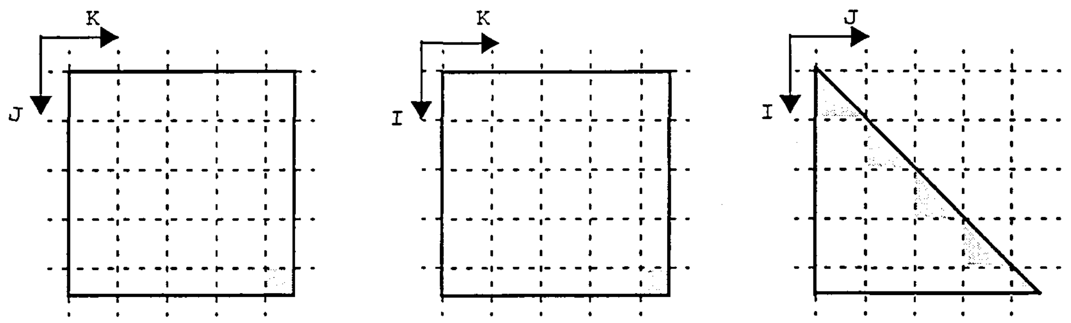


Figure 3.21: Projection of the SSYRK iteration space along (a) direction I, (b) direction J and (c) direction K.

outermost loop J. If we tile the projected iteration space at two dimensions, all tiles crossing the hypotenuse cannot be unrolled in any direction (marked in grey in Fig. 3.21c). This means that there will be a plane in the tiled code traversed by *non-unrolled loop nests*. Our goal is to reduce the number of planes of the iteration space where the boundary-tiles cannot be unrolled in any direction. Thus, in this example, we will discard loop K to be the non-tiled loop and pick loops I and J as the loops that generate less non-unrolled boundary-tiles.

To choose between loop I and J to be the non-tiled loop, our heuristic looks at their corresponding reuse vectors and selects the one that provides more temporal data locality for register reuse. In the SSYRK example, both loops carry the same amount of reuse (both reuse vectors have a weight of 1) and, therefore, we can select any of them as the non tiled loop.

Finally, once we have decided the non-tiled direction, the heuristic computes the tile sizes in proportion to the quantity of reuse carried by each tiled direction, so that the loop nest traversing core-tiles is either balanced or compute-bound and so that the amount of spill code is moderated. At this point, we want to note that using large tile sizes increases register pressure and, therefore, spill code might be generated in the loop body of the fully unrolled loop nest. However, if this increase of memory instructions due to spill-loads/ stores is less than the reduction of memory instructions due to register tiling, then it is preferable to use bigger tile sizes that generate a certain amount of spill code than to use smaller tile sizes.

We use the weight of the reuse vectors corresponding to the tiled directions to determine the ratio between the sizes in each dimension. For example, if one of the tiled directions had a weight of 2, and the other a weight of 1, we will try to pick a size for the first direction of the tile that will be twice the size of the second direction. We choose the tile sizes taking into account the number of registers needed for each reference in the loop body and the available number of machine registers as described in [23].

In the SSYRK example, and selecting loop I as the non-tiled loop, the ratio between the tile size in each dimension is 2/1. The K-direction has a reuse weight of 2, while the J-direction has a reuse weight of 1. A tile size of 4x2 needs 14 registers and a tile size of 6x3 needs 27 registers. On a 32 register machine, we would choose the 6x3 tile. At this point, we want to note that if a certain amount of ILP is achieved and register pressure is controlled then experimental results have shown that processor performance is not very sensible to the tile sizes. As an example, consider the SSYRK routine after selecting loop I as the non-tiled loop. Figure 3.22 shows the performance obtained using different tile sizes (5x3, 5x4 and 6x3). Again, measures were taken on two different processors: an ALPHA 21164 and a R10000 processor. As it can be seen, the performance obtained using different tile sizes is very similar. Moreover, the R10000 processor, due to its out-of-order nature, is even less sensible to the tile sizes than the ALPHA processor.

Summarizing, when performing the locality analysis for the register level it is important to consider the iteration space shape to decide the non-tiled loop, because the non-unrolled boundary-tiles can limit processor performance. However, it is not that important to be precise when selecting the tile sizes, if a certain amount of ILP can be guaranteed and register pressure is controlled.

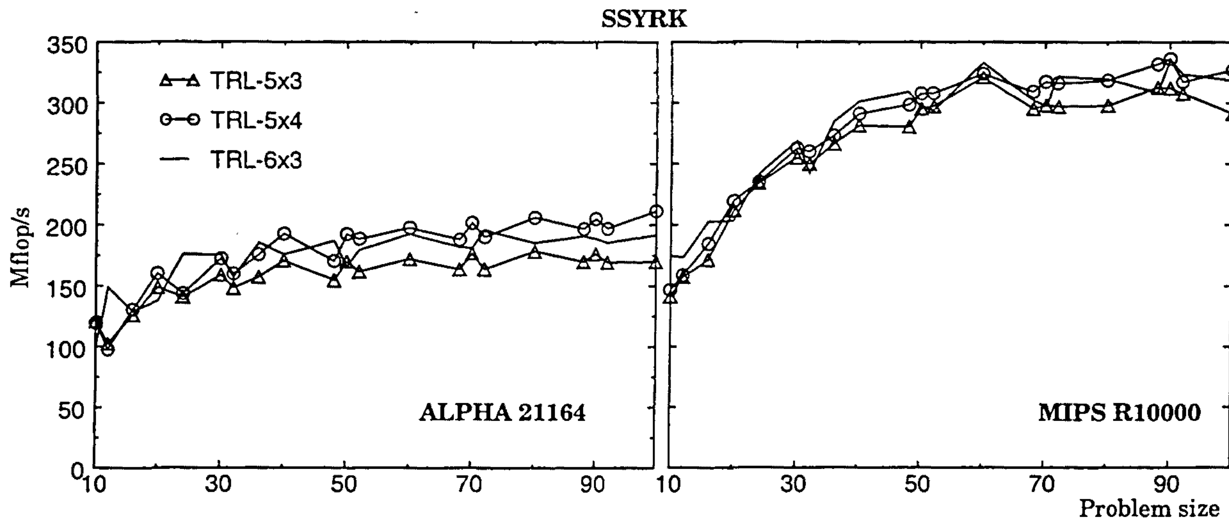


Figure 3.22: Performance of SSYRK on the ALPHA 21164 and the MIPS R10000 processor after register tiling, varying the problem size from 10 to 100 and using different tile sizes. The non-tiled loop in all cases is loop I.

3.5 PERFORMANCE EVALUATION

In this section we will present the performance results obtained by tiling for the register level, and we will compare it against the native compilers and preprocessors on two different superscalar microprocessors. 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.

3.5.1 Evaluation Process

Benchmark Programs

As benchmark programs, we have used 9 linear algebra algorithms having non-rectangular, 3-dimensional iterations spaces. Table 3.4 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 Chapter 2 (Section 2.3.3), 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 "loop limit constraints" indicates whether the dependence analyzer needs to consider the loop limit constraints to determine if the loop is fully

Ref	Name	Description	perfect nested	affine bounds	loop limit constraints
[57]	MMtri	Triangular matrix product	Yes	2	No
[122]	LU	LU decomposition without pivoting	No	2	Yes
[27]	CHOL	Cholesky factorization	No	2	Yes
[121]	QR	Givens QR-decomposition	No	2	Yes
BLAS [39]	SSYR2K	symmetric rank 2k update	Yes	1	No
	SSYRK	symmetric rank k update	Yes	1	No
	SSYMM	symmetric matrix-matrix operation	No	1	No
	STRMM	product of triangular and square matrix	Yes	1	Yes
	STRSM	solve a matrix equation	No	1	Yes

Table 3.4: Description and characteristics of several linear algebra algorithms.

permutable or not [14][51][89]. In one case (programs QR), we were forced to apply loop skewing, before tiling, to convert the loops into a fully permutable loop nest [121]. All results presented for QR were measured using the code once skewed (we note that the skewed codes obtain the same or better performance, depending on the problem size, than the non-skewed codes).

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 [98]. The non-tiled loop was selected using the heuristic proposed in Section 3.4.3 and 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. We want to remark that we do not consider the problem size when selecting the tile sizes. Hence, we use the same tile sizes for different problem sizes. However, better performance can be achieved if we consider it and select the tile sizes so that problem size becomes a multiple of the tile sizes, and thus, less boundary-tiles are executed. This fact is specially important for small problem sizes where the time wasted on boundary-tiles is a significant fraction of the total execution time.

Table 3.5 summarizes for each program the non-tiled dimension and the tile sizes selected for the register level. For each benchmark program we show: the main loop body[7] of the original loop nest

Program	Main Loop Body	Tiling Parameters at the Register Level		
		non-tiled loop	tile sizes	loop order
MMtri	C(I,J)=C(I,J)+A(I,K)*B(K,J)	K	(JxI)-(4x4)	JR-IR-K-**J-I**
LU	A(I,J)=A(I,J)-A(I,K)*A(K,J)	K	(JxI)-(4x4)	JR-IR-K-**J-I**
CHOL	A(I,J)=A(I,J)-A(I,K)*A(J,K)	K	(JxI)-(4x4)	JR-IR-K-**J-I**

QR	T1=A(J-I-1,K)
	T2=A(J-I,K)
	A(J-I-1,K)=C(I,J)*T1-S(I,J)*T2
	A(J-I,K)=S(I,J)*T1+C(I,J)*T2
		J	(IxK)-(3x6)	IR-KR-J-**I-K**

SSYR2K	C(I,J)=C(I,J)+B(J,K)*A(I,K)+A(J,K)*B(I,K)	I	(KxJ)-(4x4)	KR-JR-I-**K-J**
SSYRK	C(I,J)=C(I,J)+A(J,K)*A(I,K)	I	(KxJ)-(6x3)	KR-JR-I-**K-J**

SSYMM	C(K,J)=C(K,J)+A(K,I)*B(I,J)
	C(I,J)=C(I,J)+A(K,I)*B(K,J)
		K	(JxI)-(2x4)	JR-IR-K-**J-I**

STRMM	B(I,J)=B(I,J)+A(I,K)*B(K,J)	K	(JxI)-(4x4)	JR-IR-K-**J-I**
STRSM	B(I,J)=B(I,J)-A(I,K)*B(K,J)	K	(JxI)-(4x4)	JR-IR-K-**J-I**

Table 3.5: Non-tiled dimension, loop order and tile sizes selected for each program.

7.For the non-perfectly nested programs (LU, CHOL, QR, SSYMM and STRSM), we are not showing the additional statements outside the innermost loop.

Architecture	**MHz**	**issue rate (instr/cycle)**	**L/S per cycle**	**int/fp units**	**int/fp regs**	**L1**	**TLB entries**
ALPHA 21164	266	4 (in-order)	2/1	2/2	32/32	8KB direct mapped	64
MIPS R10000	250	4 (out-of-order)	1/1	2/2	32/32	32KB (2-way)	64

Table 3.6: Characteristics of the architectures ALPHA AXP 21164 and MIPS R10000.

(column 2), the non-tiled dimension (column 3), the tile size in each tiled dimension (column 4), and the order of the loops used in our measurements (column 5). The loops that are fully unrolled in the core-tiles are marked in bold, and `IR`, `JR` and `KR` are the iteration variables for the tile loops. Note that although the main loop bodies of different programs look very similar, they have different iteration space shapes (Appendix C shows the original code of all our benchmark programs). The iteration space shape differences force the commercial compilers to apply widely different optimizations, as we shall see later.

Target Architectures

All our measurements were taken on a uniprocessor system with an ALPHA 21164 processor [15] and on a single R10000 processor [130] of a multiprocessor system (SGI Origin 2000 [81]). The two different architectures are shortly described in Table 3.6. We will use the MFLOP/s metric as our indicator of performance and we note that operations such as DIV and SQRT, that appear in statements outside the innermost loop in programs LU, CHOL, QR and STRSM (see Appendix C), are counted as only one operation.

3.5.2 Performance Results

To compare tiling for the register level against other optimizing code transformations performed by commercial compilers and preprocessors, such as software pipelining and outer unrolling, we evaluate three different versions of each program: one is the original version (ORI) with no previously restructuring transformation. A second one is generated using the KAP[8] preprocessor [78] to restructure the code (KAP) and the third one is generated using our tool, also as a preprocessor, to tile for the register level (TRL). For the ORI version we always select the loop order that achieves, in average, the best performance and we feed the KAP preprocessor with the ORI version. It is worth noting that the best loop order for small problem sizes that fit in the cache level cannot be the best loop order for large problem sizes.

8.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. We have used version 3.1a of the KAP preprocessor.

After generating the different versions for each program, we use the standard Fortran 77 compiler[9] to generate the final executables. For the ORI and KAP versions, the F77 compiler was used with the scalar optimizations recommended by the manufacturer turned on (-O5 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 and it also perform software pipelining. For the TRL-version, however, the F77 compiler was used with the software pipelining flag turned off (-O4 on the ALPHA and -SWP:=OFF on the MIPS) because we want to isolate the benefits of tiling for the register level against other transformations, including software pipelining.

To evaluate the performance improvement of tiling for the register level, we will show the MFLOP/s obtained on both processors for the 9 benchmark programs when compiled using the different versions of the codes. On the R10000 machine, only the ORI and TRL versions are presented, since the MIPSpro compiler by itself already uses a loop nest optimizer (LNO) that performs high-level optimizations that improve program performance by exploiting ILP and caches. LNO is integrated into the compiler back end and is not a source-to-source preprocessor. We will first present results for small-to-medium problem sizes[10] (from 10 to 100), where high memory levels does not affect processor performance, and then we present results for medium-to-large problem sizes (from 100 to 1500), where cache levels and TLB harm processor performance.

Small-to-Medium Problem Sizes

Figure 3.23 shows the performance obtained on the ALPHA 21164 processor by the three versions of the programs with matrix sizes varying from 10 up to 100.

As it can be seen, tiling the register level is, in general, better than the optimizations performed by the KAP and F77 compilers for all problem sizes. Moreover, KAP performs better or equal than ORI in almost all programs (only for the MMtri program with small problem sizes (between 10 and 50) and for the SSYR2K program with medium problem sizes (between 90 and 100) KAP performs worse than ORI).

The KAP preprocessor was able to perform certain optimizations for some, but not all, program benchmarks. For example, it applies unroll-and-jam (in only one dimension) for five programs (QR, SSYR2K, SSYRK, STRMM and STRSM). It also applies other transformations such as scalar replacement, loop permutation, cache tiling and operation reordering. Operation reordering is used to help the compiler to perform an efficient instruction scheduling and KAP applies it to three programs (QR, STRMM and STRSM). Table 3.7 summarizes the transformations performed by KAP for each of the nine programs.

9.Version 4.1 of Digital Fortran on the ALPHA processor and version 7.2.1 of MIPSpro Compiler on the MIPS processor.
10.A matrix size (or problem size) of N means that we are using matrices of N by N elements.

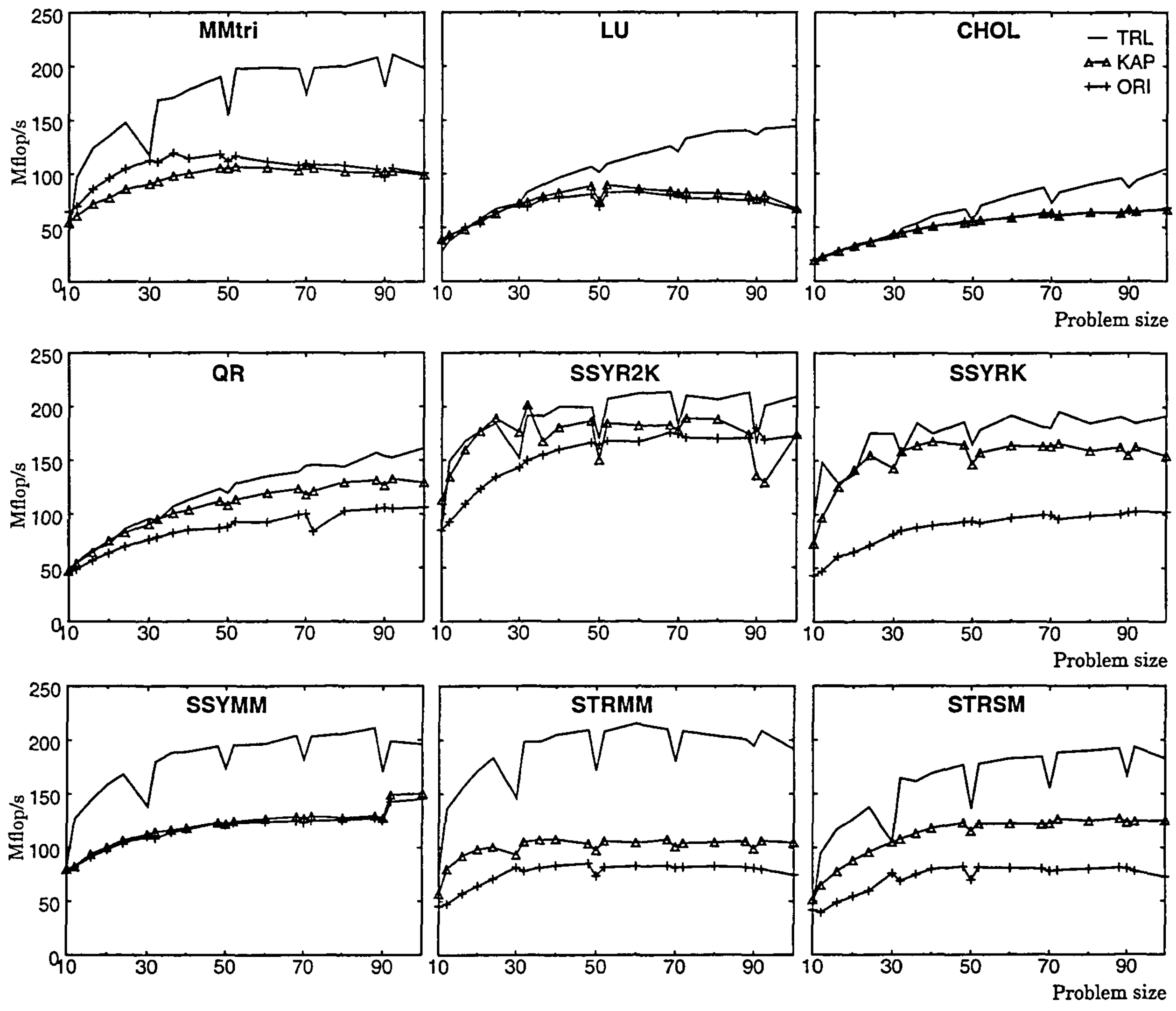


Figure 3.23: Performance obtained on the ALPHA 21164 processor by the ORI, KAP and TRL versions for the nine benchmark programs, varying the problem size from 10 to 100.

Comparing KAP with respect to ORI, it can be seen that KAP only performs better in the five programs it was able to apply unroll-and-jam (QR, SSYR2K, SSYRK, STRMM and STRSM). The performance improvement is due to a reduction in the number of load/store executed caused by the unroll-and-jam transformation. Moreover this transformation also increases ILP since the unrolled loop, in the five programs, does not carry a dependence.

Another interesting point to note is the different levels of performance obtained by KAP and ORI in programs SSYR2K, SSYRK, STRMM and STRSM despite all of them having a similar loop body (see Table 3.5). In STRMM and STRSM, less performance (compared to SSYR2K and SSYRK) is achieved because of the references B(I,J) and B(K,J) that appear in the loop body. For these programs, the

Program	Optimizing Transformations performed by KAP				
	loop permutation	scalar replacement	unroll & jam	operation reordering	cache tiling
MMtri	YES	YES	----	----	----
LU	----	----	----	----	----
CHOL	----	----	----	----	----
QR	----	----	YES	YES	----
SSYR2K	YES	YES	YES	----	YES
SSYRK	YES	YES	YES	----	YES
SSYMM	----	YES	----	----	----
STRMM	YES	----	YES	YES	----
STRSM	YES	----	YES	YES	----

Table 3.7: Transformations performed by KAP for each benchmark program.

dependence analyzer needs to consider the loop limit constraints to determine that there is not a dependence between the two references. However, we think that the dependence analyzer of the native compiler does not consider loop limit constraints and assumes that the dependence exists, thus limiting the degree of ILP that could be achieved by the software pipelining transformation.

In SSYR2K and for medium problem sizes, KAP performs worst than ORI, due to a bad exploitation of the cache level. For this program, KAP performs tiling for the cache level. However, the tiling parameters were not properly selected since the ORI version has more data locality at the cache level than the KAP version. We will see this fact more clearly in the next subsection, when dealing with large problem sizes.

For the remainder 4 programs (MMtri, LU, CHOL and SSYMM) KAP performs worse or equal than ORI. In LU and CHOL, KAP does not modify the original code. In SSYMM, it applies scalar replacement to move invariant references outside the innermost loop. However, the native compiler by itself is already capable to do this same optimization and, therefore, no performance difference is seen. Finally, in MMtri, KAP applies loop permutation to increase data locality and also adds a zero-trip[11] test to the innermost loop [129]. The execution time overhead due to the zero-trip test is significant for small problem sizes, since the innermost loop does not execute enough iterations to hide the cost of the test, thus degrading processor performance. On the other hand, the benefits of the loop permutation transformation will be slightly perceived for larger problem sizes.

11.A conditional statement to prevent execution of the innermost loop with zero-trip count.

Note that KAP was not able to apply unroll-and-jam to MMtri, LU and CHOL programs, although they have loop bodies similar to SSYR2K, SSYRK, STRMM and STRSM. The problem here is the irregularity of the iteration space shape that prevents KAP to perform unroll-and-jam. In the former programs there are two bound components in the original code that are affine functions of the surrounding loops iteration variables, while in the later there is only one (see Table 3.4).

Tiling the register level outperforms compiler optimizations such as inner unrolling, unroll-and-jam and software pipelining used in commercial compilers and preprocessors due to two main reasons: 1) it always achieves ILP in the loop body and 2) the number of load/store instructions is significantly reduced. Transformations such as inner unrolling and software pipelining can also achieve good levels of ILP in most of the cases. However, they do not exploit register reuse and thus, they do not reduce memory instructions. On the other hand, unroll-and-jam exploits data reuse at the register level, but only in one dimension of the iteration space. Register tiling, however, exploits data reuse in more than one dimension. As shown in Section 3.3, exploiting data reuse in more than one dimension achieves more data locality than exploiting data reuse in only one dimension, without increasing register pressure. This is especially important at the register level, since being able to fully exploit the (small) storage space available at this level can make a big performance difference.

In general, on the ALPHA processor, the performance improvement of TRL is much better for medium problem sizes (80-90) than for very small problem sizes (10-30) and it is also much better for problem sizes multiple of the tile sizes than for problem sizes not multiple of the tile sizes. 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 these tiles have less ILP and less data reuse than *core*-tiles. The spikes in the TRL curves that can be seen in Fig. 3.23 for almost all programs correspond to problem sizes not multiple of the tile sizes. Note that these spikes become less noticeable as the problem size increases.

In LU, CHOL, QR and STRSM programs, there is another reason that explains the small performance improvement achieved for small problem sizes. These programs perform very time consuming, non-pipelined operations (SQRT and/or DIV)[12] and only when the matrix size increases, the execution time spent in these operations can be hidden by the other operations in the loop body.

Let's now see what happens on the R10000 processor. Figure 3.24 shows the performance obtained on the R10000 processor by the ORI and TRL versions of the programs with matrix sizes varying from 10 up to 100. Recall that the MIPSpro compiler by itself already performs high-level optimizations.

The results on the R10000 processor are similar to the results on the ALPHA processor. Tiling the register level is also better than the optimizations performed by the native F77 compiler, except for very small problem sizes (between 10 and 30) and for only 4 programs (MMtri, LU, CHOL and

12.These operations appear in statements outside the innermost loop, that is, they are not in the main loop body.

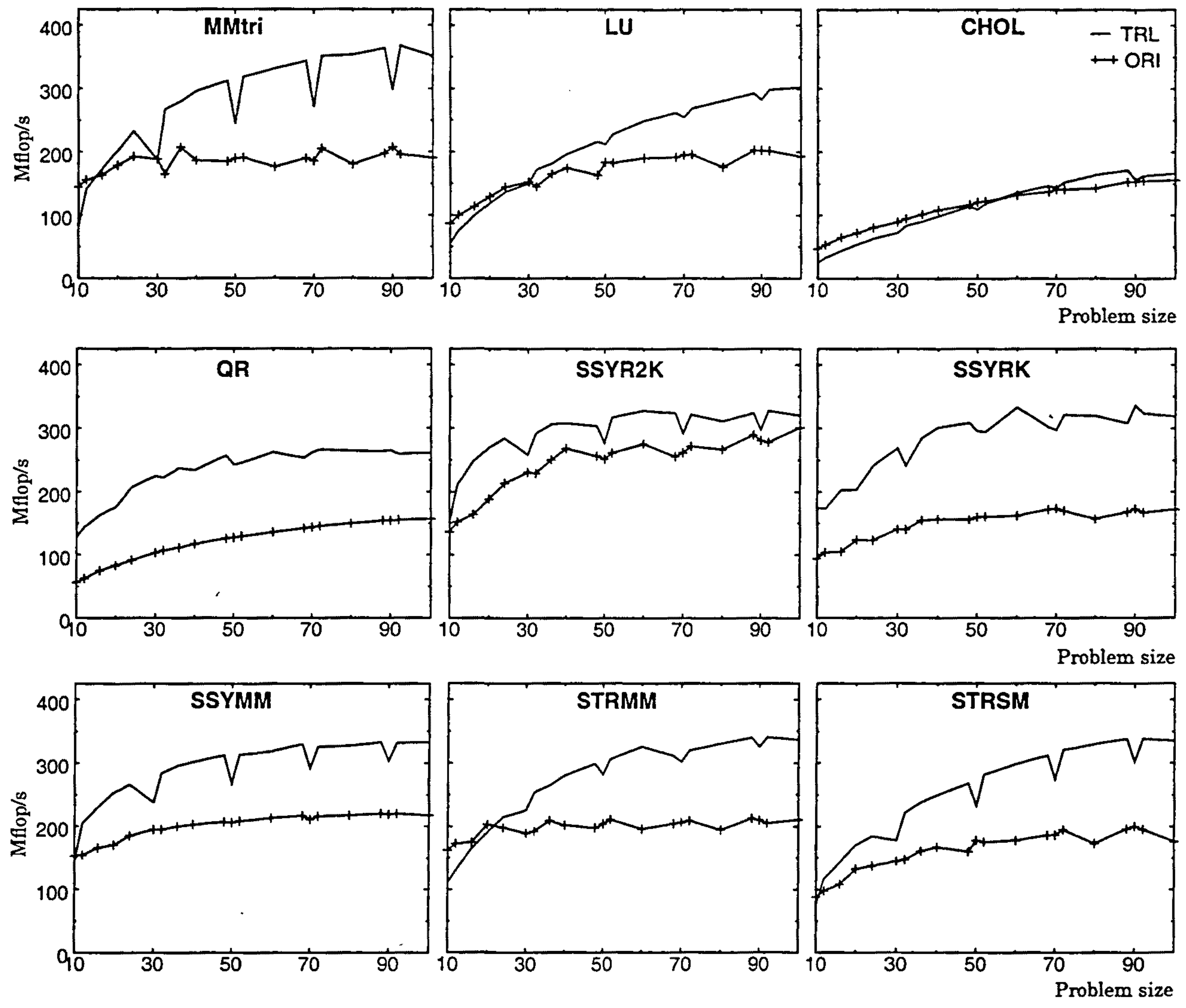


Figure 3.24: Performance obtained on the MIPS R10000 processor by the ORI and TRL versions for the nine benchmark programs, varying the problem size from 10 to 100.

STRMM). There are two reasons why ORI performs better than TRL for small problem sizes. On one hand, as already mentioned, 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 these tiles have less ILP and less data reuse than *core*-tiles. On the other hand, the TRL version uses the same tile sizes for different problem sizes. That is, we do not consider the problem size for selecting the tiling parameters despite the fact that, in our programs, the problem size is known at compile time. Note that better performance could be achieved by considering it and selecting tile sizes divisible by the problem size; doing so, less boundary-tiles would be executed. By contrast, the MIPSpro compiler does take into account the problem size when performing optimizations such as unroll-and-jam. For example, it unrolls loops by different factors depending on the problem size. This fact is specially important for small problem sizes where the time wasted on boundary-tiles is a significant fraction of the total execution time.

Program	**Optimizing Transformations performed by the MIPSpro compiler**				
	loop permutation	**scalar replacement**	**unroll & jam**	**operation reordering**	**cache tiling**
MMtri	----	Yes	Yes	----	Yes
LU	Yes	Yes	Yes	----	Yes
CHOL	----	Yes	Yes	----	----
QR	----	----	----	----	Yes
SSYR2K	----	Yes	Yes	----	Yes
SSYRK	----	Yes	Yes	----	Yes
SSYMM	----	Yes	----	----	----
STRMM	----	Yes	Yes	----	Yes
STRSM	Yes	Yes	Yes	----	Yes

Table 3.8: Transformations performed by the MIPSpro compiler for each benchmark program.

Table 3.8 summarizes the transformations performed by the MIPSpro compiler for each of the nine programs[13]. As already mentioned, the compiler performs different optimizations for different problem sizes. Basically, the main difference between codes using different problem sizes are the unrolling factors and the tile sizes selected for the cache levels (for small problem sizes, it does not perform tiling for the cache levels). In Table 3.8 we show the transformations performed for a problem size of 700.

Note that the MIPSpro compiler is able to apply unroll-and-jam to MMtri, LU and CHOL programs, while the KAP preprocessor was not. However, similarly to KAP, the MIPSpro compiler only applies unroll-and-jam in one dimension of the iteration space, and therefore, it does not exploit the register level as well as TRL. It also applies scalar replacement and cache tiling to more programs than the KAP preprocessor. Moreover, while KAP performs tiling only for the first level cache, the MIPSpro compiler also performs tiling for the second level cache in four programs (MMtri, SSYR2K, SSYRK and STRMM). However it does not perform operation reordering. Operation reordering can be a beneficial transformation in in-order processors like the ALPHA 21164, because it helps the compiler to perform a more efficient instruction scheduling. However, in out-of-order processor, like the R10000, this type of optimizations are not so important, since the out-of-order execution solves the instruction scheduling limitations.

13.To see the optimizing transformations performed by the MIPSpro compiler, we used the -FLIST:=ON option that invokes a translator, integrated into the back end stage of the compiler, that converts the compiler's internal representation into the original source language, showing the transformed code in the original source language after performing the transformations.

	MMtri	LU	CHOL	QR	SSYR2K	SSYRK	SSYMM	STRMM	STRSM
ALPHA TRL/KAP	1.6	1.2	1.1	1.1	1.1	1.2	1.5	1.8	1.3
MIPS TRL/ORI	1.3	1.1	0.8	2.0	1.2	1.8	1.4	1.2	1.4

Table 3.9: Speedups obtained by TRL over the KAP and ORI versions on the ALPHA and MIPS processors, using small-to-medium problem sizes (10-100).

Summarizing, experimental results show that transformations such as unroll-and-jam, scalar replacement and software pipelining, alone or combined, cannot achieve as high performance as register tiling. In Table 3.9 we have summarized the speedups of TRL over KAP (for the ALPHA processor) and over ORI (for the MIPS processor) for small problem sizes. For each program version the harmonic mean of the MFLOP/s obtained for different matrix sizes is computed (we used 21 different problem sizes, going from 10 to 100). Then, we compute the speedup of TRL over KAP and ORI versions by dividing these harmonic means. On the ALPHA, the speedups over the KAP preprocessor are in the range 1.1 to 1.8 and on the MIPS, the speedups over the MIPSpro compiler, vary between 0.8 and 2.0.

Medium-to-Large Problem Sizes

Now, we will show the behavior of TRL for medium-to-large problem sizes where high memory levels, such as cache and TLB, might negatively affect performance. Figure 3.25 shows the performance obtained on the ALPHA 21164 processor by the three versions of the programs with matrix sizes varying from 100 up to 1500.

As expected, the performance obtained by the three versions of the programs (TRL, KAP and ORI) decreases with large problem sizes. However, it can be seen that TRL performs better than the other versions, 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.

Another point to notice is the performance obtained by KAP in the SSYR2K program. As shown in Table 3.7, KAP performs tiling for the cache level in the SSYR2K and SSYRK programs. While in the SSYRK program, KAP performs better than ORI, the same does not happen for SSYR2K. We believe the reason behind this performance degradation is the tile size and loop ordering chosen by KAP, which happen to decrease performance rather than increase it.

Finally, we note that in 6 programs (MMtri, LU, CHOL, QR, STRMM and STRSM) the performance of TRL decreases faster than in SSYR2K, SSYRK and SSYMM. In these 6 programs, there is at least one array reference that is traversed in row major order by the non-tiled loop. Thus, for these

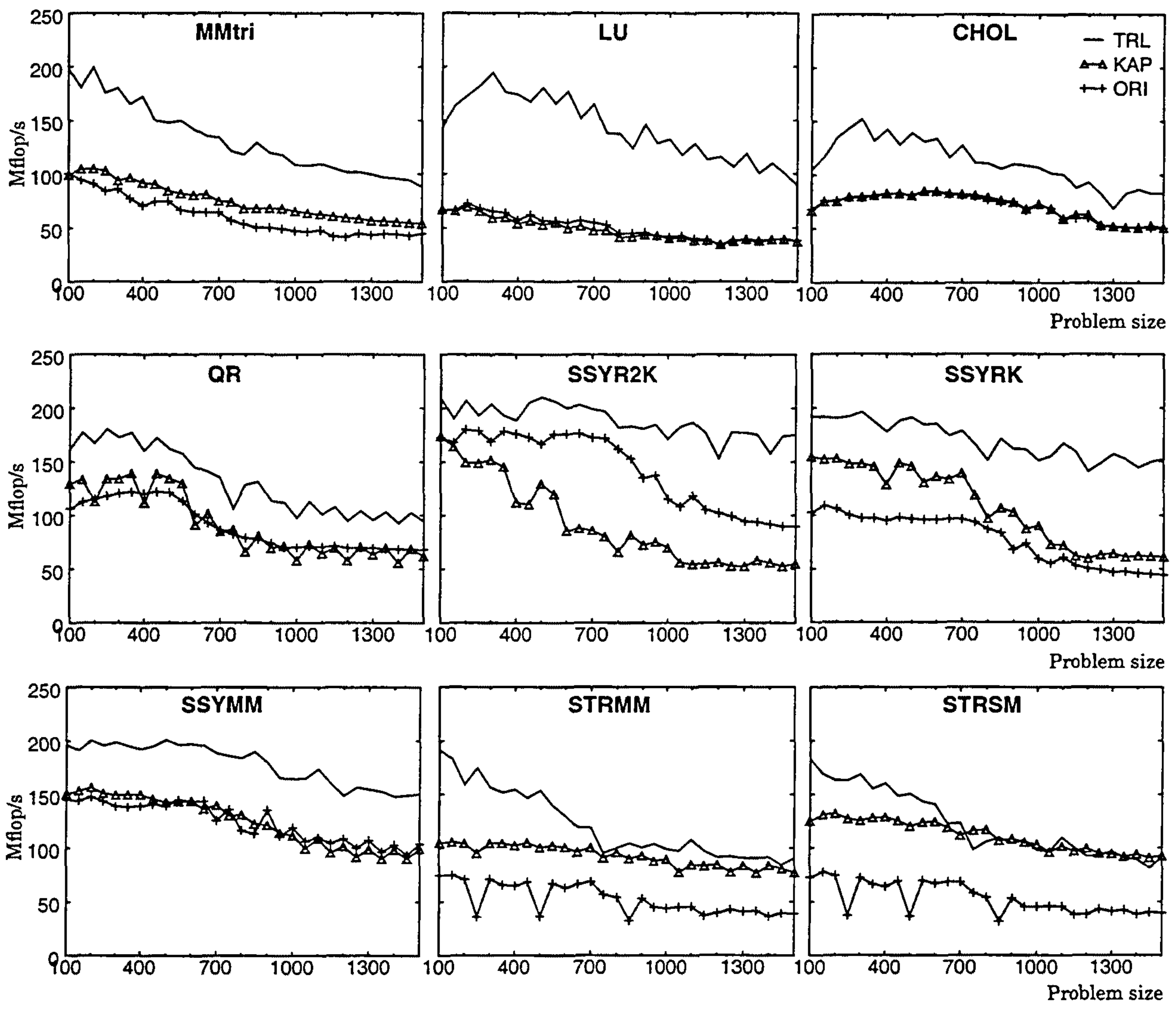


Figure 3.25: Performance obtained on the ALPHA 21164 processor by the ORI, KAP and TRL versions for the nine benchmark programs, varying the problem size from 100 to 1500.

references spatial locality is not being exploited[14], and TLB misses increase considerably. For large problem sizes, tiling only for the register level can substantially increase TLB misses when spatial locality is not being exploited. However, we will see in Chapter 5 that this problem can be solved by performing tiling also for higher levels of the memory hierarchy.

Results for the R10000 processors.are shown in Fig. 3.26. Note that the behavior of all programs on the MIPS processor is, more or less, the same as on the ALPHA processor. However, we note that on the MIPS processor, performance begins to decrease at much larger problem sizes, because the MIPS processor has a first level cache four times bigger than the ALPHA processor. Moreover, MIPS's cache is two way associative while ALPHA's cache is direct mapped, thus reducing conflict misses.

14.Programs are written in FORTRAN, that stores matrices in column major order.

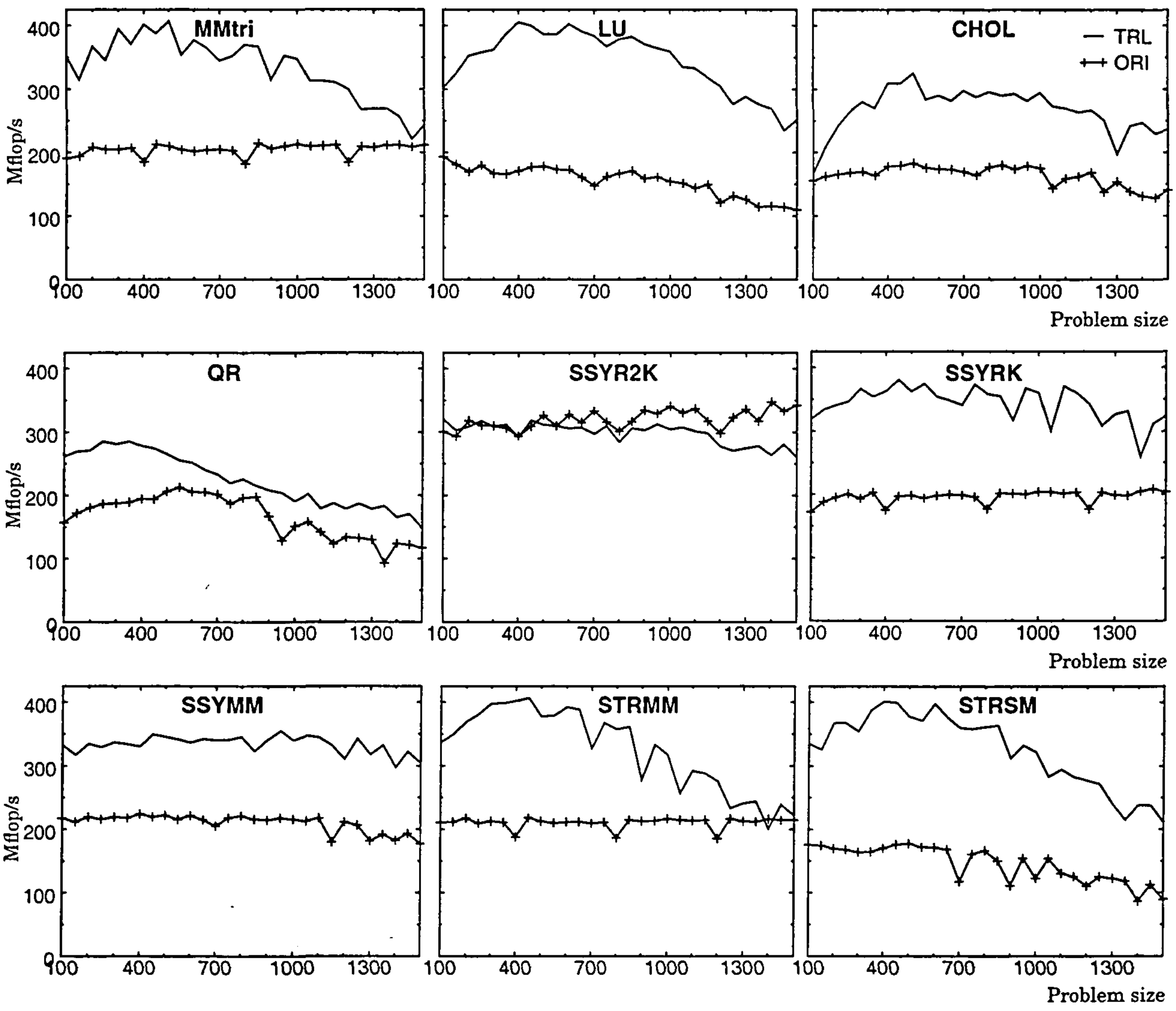


Figure 3.26: Performance obtained on the MIPS R10000 processor by the ORI and TRL versions for nine the benchmark programs, varying the problem size from 100 to 1500.

Notice also that ORI performs better than TRL in SSYR2K. In this program (and also in MMtri, SSYRK and STRMM), the MIPSpro compiler not only performs tiling for the first level cache, but also for the second level, thus achieving a stable performance for all problem sizes. TRL, however, do not exploit data locality for higher levels of the memory hierarchy and thus, performance decreases. We note that in MMtri, SSYRK and STRMM programs and for problem size larger than 1500, ORI also performs better than TRL.

Summarizing, results show that tiling for the register level also achieves good performance for medium-to-large problem sizes, despite the fact that tiling for higher levels of the memory hierarchy has not been performed. However, tiling only for the register level must be applied with care since it

	MMtri	LU	CHOL	QR	SSYR2K	SSYRK	SSYMM	STRMM	STRSM
ALPHA TRL/KAP	1.7	2.9	1.5	1.5	2.4	1.8	1.5	1.2	1.1
MIPS TRL/ORI	1.6	2.2	1.6	1.4	0.9	1.7	1.6	1.5	2.3

Table 3.10: Speedups obtained by TRL over the KAP and ORI versions on the ALPHA and MIPS processors, using medium-large problem sizes (100-1500).

can heavily increase the overall TLB misses if spatial locality is not properly exploited, resulting in a performance degradation. Table 3.10 summarizes the speedups of TRL over KAP (for the ALPHA processor) and over ORI (for the MIPS processor) for large problem sizes. As before, we used the harmonic mean of the MFLOP/s obtained for different matrix sizes (we used 29 different problem sizes, going from 100 to 1500) to compute the speedups. On the ALPHA, the speedups over the KAP preprocessor are in the range 1.1 to 2.9 and on the MIPS, the speedups over the MIPSpro compiler, vary between 0.9 and 2.3.

Cost of Performing ISS

We conclude this section by presenting some experimental data on the number of times that ISS has been performed and the number of loop nests generated for our 9 benchmark programs.

As shown in Section 3.2.4, the number of times that our algorithm performs ISS (and also the number of loop nests generated) depends on the number of bound components of the UCLs in the tiled code (just, after the iteration space tiling phase). Thus, we have divided the 9 benchmark programs into three groups according to the number of loop bound components of the UCLs after the iteration space tiling phase. Programs MMtri, LU, SSYMM, STRMM and STRSM belong to group A, QR, SSYR2K and SSYRK belong to group B and CHOL forms group C.

Since the number of bound components of the UCLs increment with the number of memory levels being exploited, we will present the cost of applying our method when tiling has been applied only to the register level and when it has been applied to both the cache and register levels. At each level, two dimensions of the 3-dimensional iteration spaces were tiled.

Table 3.11 presents the results for each group. Columns 2 to 8 shows data when tiling has been applied only at the register level. Columns 2 and 3 (labeled "R" and "S+M", respectively) indicate the number (and type) of loop bound components of the UCLs in the tiled code (recall the notation used in Section 3.2.4). Column 4 (labeled "N_{iss}") indicates the number of times that ISS has been performed. We note that, for all programs, the actual number of ISSs performed is exactly the minimum number of necessary ISSs (as computed using the formula of Table 3.2 on page 89). Column 5 (labeled

Program Group	Tiling only register level							Tiling cache and register levels						
	UCL's bound components		N_{iss}	N_{loop_nests}	UCLs unrolled			UCL bound components		N_{iss}	N_{loop_nests}	UCLs unrolled		
	R	S+M			2	1	0	R	S+M			2	1	0
A	2	2+0	8	9	1	4	4	2	4+0	14	15	1	6	8
B	2	2+0	8	9	1	4	4	4	2+0	14	15	1	6	8
C	3	1+1	8	16	1	4	11	4	3+1	20	40	1	7	32

Table 3.11: Number of loop bound components, number of times that ISS has been performed and number of loop nests generated for each benchmark group, when tiling has been applied only to the register level, and when it has been applied to both the cache and register levels.

"N_{loop_nests}") gives the total number of loop nests generated at the end of the process and columns 6, 7 and 8 indicate how many of them are *fully-unrolled loop nests*, *partially-unrolled loop nests* or *non-unrolled loop nests*, respectively. Columns 9 to 15 are the same as columns 2 to 8 but, in this case, tiling has been applied to both the cache and register level. Note that when two levels are tiled (cache and registers) the number of bounds in the UCLs is greater and, therefore, the final number of loop nests generated increases.

Finally, we want to indicate that tiling for the register level increases code size and this fact could increase the instruction cache misses. However, on the ALPHA processor, we have seen by instrumenting the executables with the ATOM tool [36], that the overall instruction cache misses is insignificant for all benchmark programs. The reason is that the generated code has a good degree of locality; in average for all programs, 97% of the referenced instructions are done by only 10% of the executed code. Register tiling 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 (obviously, the *fully-unrolled loop nest* that traverses the *core*-tiles is the one that is executed the most).

3.6 RELATED WORK

There has been much discussion in the literature regarding memory hierarchy management [23][25][27][98][122], but it has mostly focused on exploiting data reuse for the cache level and little attention has been paid to the register level.

M. Wolf in [121] 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 [23] and [24] uses unroll-and-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 [23] and [24] by allowing several inner loops to have affine bounds of multiple tiled loops iteration 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-and-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-and-jam and register tiling generate different transformed codes for non-rectangular spaces. In particular, unroll-and-jam generates more *boundary* tiles than our proposed method.

Wolf, Maydan and Chen in [124] developed an algorithm that combines tiling for the cache level, unroll-and-jam and software pipelining in an attempt to select a set of transformations that lead to high performance. They handle non-perfectly nested loops and loops with non-rectangular iteration spaces, but they do not give details in [124]. To exploit the register level, they use unroll-and-jam as described by Carr and, as mentioned before, it can only be applied in limited cases of non-rectangular iteration spaces.

There has been also much work regarding locality analysis, but, again, it is mostly focused on the cache level. K. Kennedy and K. Kinley in [69] propose an easy algorithm to determine a good order of the loops that exploits locality at the cache level, but they don't explicitly indicate which loops could be tiled. They also need to know the number of iterations of each loop. Usually the number of iterations of each loop is unknown either because it is an input parameter to the program or because the iteration space is non-rectangular and the number of iterations varies each time it is executed. M. Wolf and M. Lam in [123] propose an algorithm that determines which loops carry reuse at the cache level and then, they tile all of them. They do not provide a specific ordering of the loops, since they assume that all data referenced inside a tile will fit into the memory level being exploited. The implication of this assumption is that their tiles must be smaller and, therefore, cannot take advantage of the full memory size. Being able to use as much as possible of a memory level capacity is very important at the register level because the number of registers is small. In Section 3.4 we have seen that, by properly establishing an ordering among the loops, it is not necessary to keep all data in the memory level being exploited and, therefore, we can have bigger tile sizes. Finally, S. Carr in [23] proposes an heuristic to determine which loops should be unrolled and jammed and the unrolling factors. He considers the amount of reuse carried by each loop, but does not take into account the iteration space shape. However, as shown in Section 3.4.2, for small problem sizes and/or very irregular problems, it is important to consider the iteration space shape for reducing the time wasted executing *boundary*-tiles.

Finally, we note that current commercial compilers and preprocessors are not always 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.

3.7 SUMMARY

Tiling for the register level in more than one dimension of the iteration space 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 chapter 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 divides first the original iteration space into regular tiles, using the strip-mining and loop permutation transformations. 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 *core*-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.

We have also proposed a simple heuristic that determines the loops to be tiled at the register level. Our heuristic considers not only temporal reuse, but also the iteration space shape. Moreover, the heuristic is simple enough to be suitable for automatic implementation by compilers.

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 general, our method has outperformed the native compilers and the KAP preprocessor, showing speedups up to 2.9.