A Study of Loop Nest Structures and Locality in Scientific Programs

by

Robert Sawaya

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

A Study of Loop Nest Structures and Locality in Scientific Programs

Robert Sawaya

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

1998

This study evaluates four techniques that improve the structures of loop nests, in order to make transformations targeting perfect loop nests more applicable. These techniques are code sinking, loop distribution, loop distribution with scalar expansion, and loop fusion. This study also exmines the subscript expressions of array references, in order to conclude whether spatial locality can be enhanced. This research is conducted on 23 applications from the Perfect Club, Nas, Spec92 and NCSA benchmark suites.

The results indicate that code sinking and loop distribution— with or without scalar expansion— can be effective in increasing perfect nests, in more than half of the benchmark applications. The overhead introduced by loop distribution is negligible, while the overheads of code sinking and especially of loop distribution with scalar expansion are found to be prohibitive for some applications.

Furthermore, in more than half of the benchmark applications, it is found that the array references exploit locality effectively. In only a quarter of the applications, loop permutation may be beneficial to spatial locality. Similarly, flexible data layout may be beneficial in only a quarter of the applications.

Acknowledgements

First of all, I want to express my gratitude to my supervisor, Dr. Tarek S. Abdelrahman. During the past two years, he provided me with valuable guidance and priceless advice.

I would also like to thank the members of my examination committee, Dr. Corinna G. Lee, Dr. Charles Clarke and Dr. Wai Tung Ng for their valuable comments on my thesis.

I am thankful for the financial help during the past two years, provided through a Graduate Scholarship from the Natural Sciences and Engineering Research Council of Canada.

I would also like to thank my parents, Georges and Khaoula, my sister Carole, and my brothers Lou, Maurice and Edward for their care, support and love.

To my fiancée, Faten, words cannot describe the gratefulness I feel for the support, strength and love you have given me.

Finally, I dedicate all my work to my very special friend, Shamita Mukherjee. I will never forget her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Scientific applications motivated the early development and accelerated the advances of computers. Today, due to their characteristics, scientific applications continue to drive advances in high-performance computing. In fact, some scientific applications, called data-parallel, require multi-processors to provide the needed processing power. Data-parallel applications consist, mainly, of loops iterating over a set of large arrays.

In order to achieve high-performance for data-parallel applications, optimizing and parallelizing compilers are highly desirable. Most data-parallel applications are written as sequential programs. Thus, compilers that can extract parallelism from sequential programs relieve application developers from the task of re-writing their codes to explicitly express parallelism. Furthermore, a high-performance compiler can relieve the developer from implementing machine-dependent source-level optimizations, hence improving portability across platforms.

As the gap between processor and memory speeds continues to widen, cache locality optimizations, in particular, are becoming a crucial responsibility of high-performance compilers [WL91a]. Indeed, many transformations and algorithms are designed to improve cache locality[IT88, LRW91, MT96]. This thesis is mainly concerned with improving loop nest structures to make these transformations more applicable.

1.1 Summary of the Study

1.1.1 Motivation

On the one hand, most cache locality optimizations, such as loop permutation and tiling, are designed to target the perfect nests of an application, i.e. nests that consist of loops with no intervening code between their headers, and no intervening code between their tails. Therefore, the applicability and effectiveness of such optimizations are limited by the relative contribution to execution time of perfect nests. This work determines how common perfect nests are, and to what extent they contribute to execution time. Moreover, this research evaluates techniques that can be used to transform imperfect nests into perfect ones.

On the other hand, many algorithms [WL91b, WL91a, KM92, CMT94, KRC97] attempt to improve spatial reuse and translate it into spatial locality. The results presented [WL91b, WL91a, KM92, CMT94, KRC97], however, show little benefit when these algorithms are applied to complete applications. The main explanation offered is that the applications already have very high hit rates (close to 95%). Other researchers [CMT94, MT96] have found that programs are usually written with good spatial locality. This research tries to verify this finding and to determine if the good locality is due to coding that effectively exploits spatial locality, but uses an approach that is independent of machine specific parameters.

1.1.2 Goals

The study has two main goals. First, the thesis evaluates techniques used to improve nest structures. It also uses the nests structures and characteristics to measure an upper bound on the expected benefits of tiling and the applicability of loop permutation. Second, the thesis studies array references and their subscript expressions, as well as loop ordering in loop nests to determine whether spatial locality can be improved by loop permutation or data layout.

1.1.3 Summary of the Results

This research establishes that on the average, perfect nests contribute to 39% of the execution time of 23 applications from 4 benchmark suites. Four transformations are evaluated for their effectiveness in increasing the relative contribution to execution time of perfect nests: code sinking, loop distribution, loop distribution with scalar expansion, and loop fusion. It is found that code sinking causes the largest increase in the relative contribution to execution time of perfect nests (56%), followed by loop distribution with scalar expansion (51%), and loop distribution (45%). Loop fusion is found to result in no significant improvement on the relative contribution to execution time of perfect nests.

Loop permutation is found to have a limited applicability across the benchmarks, mainly because the contribution to execution time of perfect nests is small. Tiling is found beneficial in few applications, because most perfect nests do not carry temporal locality. Also, it is found that increasing the contribution of perfect nests makes loop permutation more applicable and increases the potential benefit of tiling. This study also shows that loop permutation or data layout would only enhance spatial locality in a quarter of the benchmarks.

1.2 Contributions

The contributions of this research are summarized in the following points.

- The study characterizes loop nests structures in terms of the applicability of many locality optimization transformations.
- The study evaluates the benefits and overheads of four techniques that improve loop nests structures.
- The study implements two simple separate frameworks, independent of cache parameters, to assess the need for loop permutation and flexible data layout to improve spatial locality.

- This research adopts a new approach to assess the contribution of loop nests. The majority of previous studies have relied solely on the number of loop nests, which may be misleading because loop nests might have very different impacts on the overall execution time. Consequently, measurements of actual execution times are used to statistically weight the loop nests in a given application.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on high-performance compilers. Chapter 3 examines the four techniques that improve nests structures. Chapter 3 presents, also, the frameworks used to assess the need for spatial locality enhancements. Chapters 4 and 5 present the reuslts on the structure of nests and spatial locality, respectively. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Background

This chapter provides background material on loop nests and on locality enhancement transformations. It also describes past research related to this study.

2.1 Loops and Loop Nests

In this section, the issues of what constitutes a *loop nest* and what is a *perfect loop nest* are defined and illustrated. Since this research evaluates the effects of various transformations on the perfect nests of an application, it is important to clearly define what constitutes a loop nest and what makes it a perfect nest. Before discussing loop nests, some notions concerning loops need to be defined.

2.1.1 Loops

A DO-loop (later referred to as loop) is defined as a structured program construct consisting of a loop header statement, a sequence of inner statements called loop body, and a loop tail statement that marks the end of the loop [Man97].

```
do $i = i_b,\ i_f,\ s$                $\Longleftarrow$ loop header
    $<statements>(i)$             $\Longleftarrow$ loop body
end do                            $\Longleftarrow$ loop tail
```

In the above definition, i is the loop index variable, and i_b, i_f, s are integer-valued expressions, called lower bound, upper bound and step, respectively. At the loop entry, the index variable takes the value of i_b. With each iteration of the loop, the index variable is incremented by s. The loop is exited when the index variable is outside the i_b to i_f range. The loop body consists of statements in which the index variable i may appear. Therefore, there are instances of the body of the loop, depending on the value of i.

The following characteristics of loops can be defined:

- A loop A is said to *enclose* a loop B, if and only if all the statements of loop B are also statements of loop A. Loop A is said to be the *outer* loop and loop B is said to be the *inner* loop.
- The *depth level* of a loop is defined as the number of its outer loops.
- A loop A is said to *directly enclose* loop B, if and only if loop A encloses loop B and the depth level of loop A is one less than the depth level of loop B. In other words, there is no third loop C that encloses loop B and is enclosed by loop A.
- A loop is said to be a *single* loop if it encloses no loops and is enclosed by no loops.
- Two loops A and B, where A encloses B, are said to be *tightly* or *perfectly nested* if and only if loop A contains no other statement than loop B.
- Two loops are said to be *adjacent* if and only if there are no statements between the tail statement of one and the header statement of the other.

Figure 2.1 illustrates the above definitions. For instance, L_1 encloses L_2, L_3, L_4 and L_5. However, L_1 directly encloses L_2 only. L_2 directly encloses both L_3 and L_4. L_4 directly encloses L_5, while L_3 encloses no loops. Moreover, in Figure 2.1, loops L_4 and L_5 are tightly nested while loops L_1 and L_2 are not. Finally, the loops L_3 and L_4 are adjacent.

2.1.2 Loop Nests

A *loop nest* N is a sequence of loops $(L_1, L_2, ..., L_n)$, where $n \geq 1$, and where the following conditions hold:

1. Every loop L_i, where $(1 \leq i <$ n), directly encloses loop L_{i+1} only.

```
L1 do i = 1, 10, 1
      st1
L2    do j = 1, 100, 1
         st2
L3       do k = 1, 100, 1
            st4
         enddo
L4       do k = 1, 100, 1
L5          do m = 1, 100, 1
               st5
            enddo
         enddo
      enddo
      st3
   enddo
```

Figure 2.1: Example illustrating loop definitions.

2. Loop L_n may or may not enclose another loop.

Loop L_1 which is not enclosed by any loop in the nest is called the *outermost* loop of the nest. Loop L_n, which encloses no other loop belonging to the nest $(L_1, L_2, ..., L_n)$, is called the *innermost* loop of the nest. The *dimension* of the nest is the number of loops in the nest, i.e. n. In Figure 2.1 there are 2 loop nests: (L_1,L_2) and (L_4,L_5).

A loop nest is said to be *real perfect* if there is no intervening code between the headers of the loops, there is no intervening code between the tails of the loops, and the innermost loop does not enclose any loop. More rigorously, a loop nest $(L_1, L_2, ..., L_n)$ is considered real perfect if the following conditions are satisfied:

1. The outermost loop L_1 is not enclosed by a loop L_x such that L_x and L_1 are tightly nested.

2. All loops L_i and L_{i+1}, where $(1 \leq i < n)$, are tightly nested.

3. The innermost loop L_n does not enclose any other loop.

Second, a loop nest is said to be *pseudo perfect* if there is no intervening code between the headers of the loops and between the tails of the loops, and the innermost loop does not tightly enclose any loop. More rigorously, a loop nest $(L_1, L_2, ..., L_n)$ is considered pseudo perfect if the following conditions are satisfied:

1. The outermost loop L_1 is not enclosed by a loop L_x such that L_x and L_1 are tightly nested.

2. All loops L_i and L_{i+1}, where $(1 \leq i < n)$, are tightly nested.

3. The innermost loop L_n may enclose other loops but not *tightly*.

Hence , by definition, a real perfect nest is also a pseudo perfect nest.

Figure 2.2 illustrates real and pseudo perfect nests. The following remarks try to clarify the differences and similarities between the two types of perfect nests.

- According to the definition of real and pseudo perfect nests above, the code segment in Figure 2.2 contains two perfect nests: a pseudo perfect nest (L_1, L_2) and a real perfect nest (L_3, L_4, L_5).

- The first condition for both real and pseudo perfect nests avoids redundant perfect nests. For instance, the nest (L_4, L_5) is not considered a perfect nest because L_3 and L_4 are tightly nested. It is desirable to consider loop nests with the largest granularity for the purposes of loop optimizations.

- Similar to the first condition, the last condition for pseudo perfect nests, avoids redundant perfect nests. The nest (L_3, L_4) is not considered pseudo perfect because L_4, the innermost loop, encloses L_5 tightly.

- Transforming pseudo perfect nests into real perfect nests is desirable because:

  - The number of loops that can be part of an algorithm or an optimization (e.g. loop interchange) is increased.

  - The innermost loop of a pseudo perfect nest would enclose at least a loop L that is not part of the nest. The loop L, which is directly related to the spatial locality, will be shielded from the effect of a given transformation. By transforming the loop nest into a real perfect nest, the loop L becomes part of the perfect nest and thus can be affected by the optimization.

```
L1 do i = 1, 10, 1
L2     do j = 1, 100, 1
           st2
L3         do k = 1, 100, 1
L4             do m = 1, 100, 1
L5                 do n = 1, 100, 1
                       st5
                   enddo
               enddo
           enddo
       enddo
   enddo
```

Figure 2.2: There are 2 perfect loop nests. First, the nest $N_1(L_1, L_2)$ is a pseudo perfect nest. It is not real perfect because loop L_2 encloses loop L_3. Second, the nest (L_3, L_4, L_5) is a real perfect nest.

2.2 Data Dependence

Compilers rely on data dependence analysis [GKT91, MHL91, Pug92] to determine the constraints on the order of execution of the statements in a program. A data dependence exists between two statements S_1 and S_2, where S_1 is executed before S_2, that access the same memory location. There are four data dependence types:

- *True* or *flow* dependence occurs when S_1 writes to a memory location that is later read by S_2. This dependence is denoted as $S_1 \delta^f S_2$.
- *Anti* dependence occurs when S_1 reads a memory location that is later written by S_2. This dependence is denoted as $S_1 \delta^a S_2$.
- *Output* dependence occurs when S_1 writes to a memory location that is later written by S_2. This dependence is denoted as $S_1 \delta^o S_2$.
- *Input* dependence occurs when S_1 reads to a memory location that is later read by S_2. This dependence is denoted as $S_1 \delta^i S_2$.

For a data dependence, the statement that first accesses the memory location is called the *source* of the dependence, while the statement that last accesses the memory location

```
S1: X = Z-Y
S2: V = X+4
S3: X = Z-5
```

Figure 2.3: The four basic dependence types in a straight-line code

is called the *sink* or *target* of the dependence. Figure 2.3 illustrates the four types of dependence. First, there is $S_1\delta^f S_2$ because S_1 writes the variable `X` which is later read by S_2. Second, there is $S_2\delta^a S_3$ because S_2 reads the variable `X` which is later written by S_3. Third, there is $S_1\delta^o S_3$ because S_1 writes the variable `X` which is later written by S_3. Finally, there is $S_1\delta^i S_3$ because S_1 reads the variable `Z` which is later read by S_3.

2.3 Data Dependence in Loops

Extending the notion of data dependence to loops is complicated because there are multiple instances of the statements in the loops. Depending on the iterations of the source and target instances, two types of loop dependences are distinguished [BGS94].

1. A *loop carried dependence* is a dependence between two statement instances in two different iterations of a loop.

2. A *loop independent dependence* is a dependence between two statement instances in the same iteration of a loop.

Each statement in a loop might be executed more than once, thus the need for distinguishing instances of statement executions becomes necessary. Given a statement S_1 enclosed in a loop, such as in Figure 2.4, the instance of S_1 for iteration $i = i_1$ is labelled $S_1(i_1)$. This extension to the statement notation is required to represent dependences between instances of statements executed at different iterations of the loop. For example, in Figure 2.4 the following dependences, among others, are present.

- In each iteration `i`, the statement S_1 uses and then defines `a(i)` resulting in the $S_1(i)\delta^a S_1(i)$ loop independent dependence.

```
$L_1$ do i = 1, 10, 1
$S_1$     a(i) = a(i) + b(i+1) + 7
$S_2$     b(i) = a(i-1)
$E_1$ enddo
```

Figure 2.4: Example of loop dependences.

- In the iteration `i`, the assignment to the array reference `b(i)` in S_2 writes to the i^{th} element of the array `b`. In the iteration `i-1`, the use of the array reference `b(i+1)` in S_1 reads the i^{th} element of the array `b`. Consequently, there is a loop carried dependence represented by $S_1(i-1)\delta^a S_2(i)$.

- In the iteration `i`, the assignment to the array reference `a(i)` in S_1 writes to the i^{th} element of the array `a`. In the iteration `i+1`, the use of the array reference `a(i-1)` in S_2 reads the i^{th} element of the array `a`. Consequently, there is a loop carried dependence represented by $S_1(i)\delta^f S_2(i+1)$.

- There is a loop carried dependence $S_1(i)\delta^i S_2(i+1)$ due to the array references `a(i)` and `a(i-1)` in S_1 and S_2, respectively.

2.3.1 Dependence Distance Vectors

An iteration of a loop nest with a dimension (n $\geq$ 1) is represented by an n-tuple called *iteration vector* $\mathbf{i}$. Each of the entries in $\mathbf{i}$ corresponds to the value of the index of one of the loops in the nest. The first and last entries in $\mathbf{i}$ corresponds to the outermost and innermost loops of the nest, respectively. For example, Figure 2.5 shows a nest N with dimension equal to 3. The iteration `i=2`, `j=5` and `k=3` is represented by the vector `(2,5,3)`.

Data dependences in nests are represented, if possible, using dependence distance vectors [BGS94, WL91b]. A dependence with the sink statement at iteration vector $\mathbf{i}_{sink}$ and the source statement at iteration vector $\mathbf{i}_{source}$, is represented by a dependence distance vector $\mathbf{d} = \mathbf{i}_{sink}$ - $\mathbf{i}_{source}$.

```
$L_1$ do i = 1, 10, 1
$L_2$     do j = 1, 10, 1
$L_3$         do k = 1, 10, 1
$S_1$             a(k,j,i) = a(k,j,i) + b(k+1,j-1,i+1) + 7
$S_2$             b(k-1,j+3,i-2) = a(k-2,j+2,i)
$E_3$         enddo
$E_2$     enddo
$E_1$ enddo
```

Figure 2.5: Example to illustrate distance and direction vectors.

A distance vector **d**=(d$_1$,d$_2$, ... ,d$_n$) is said to be *lexicographically positive* if and only if for any d$_i$<0 there exist a d$_j$>0 where j<i.

Figure 2.5 illustrates dependence distance vectors. The dependences, in the nest depicted in this figure are discussed below:

- The statement S_1 writes and the statement S_2 later reads the same element of the array a at iteration vectors (k,j,i) and (k+2,j-2,i), respectively. Thus, there is a loop carried flow[1] dependence from S_1 to S_2. The distance vector of this dependence is **d**=(2,-2,0). The dependence is denoted by $S_1\delta^f_{(2,-2,0)}S_2$.
- The statement S_1 reads and the statement S_2 later writes the same element of the array b at iteration vectors (k-1,j+1,i-1) and (k+1,j-3,i+2), respectively. Thus, there is a loop carried anti-dependence denoted by $S_1\delta^a_{(2,-4,3)}S_2$.
- The statement S_1 reads then writes the same element of the array a. Thus, there is a loop independent anti-dependence denoted by $S_1\delta^a_{(0,0,0)}S_1$. All the elements in the distance vector are equal to zero because the source and sink of the dependence are in the same iteration.

2.3.2 Dependence Direction Vectors

The dependence *direction vector* [BGS94, WL91b] represents an ordering between the iteration vectors of the source and the target of the dependence. More precisely, for a

[1]It is a flow dependence because the write occurs before the read since the iteration (k,j,i) occurs before (k+2,j-2,i).

dependence where the source and target iterations are represented by $\mathtt{s}=(\mathtt{s}_1,\mathtt{s}_2,\ \ldots\ ,\mathtt{s}_n)$ and $\mathtt{t}=(\mathtt{t}_1,\mathtt{t}_2,\ \ldots\ ,\mathtt{t}_n)$, respectively, the direction vector is given by $\mathbf{d}=(\mathtt{d}_1,\mathtt{d}_2,\ \ldots\ ,\mathtt{d}_n)$ where, for $(1 \leq \mathrm{i} \leq \mathrm{n})$,

$$\mathtt{d}_i = \begin{cases} < & \text{if } \mathrm{s}_i < \mathrm{t}_i \\ = & \text{if } s_i = \mathrm{t}_i \\ > & \text{if } \mathrm{s}_i > \mathrm{t}_i \end{cases}$$

In Figure 2.5, there are the following dependences with direction vectors: $S_1\delta^f_{(<,>,=)}S_2$, $S_1\delta^a_{(<,>,<)}S_2$ and $S_1\delta^a_{(=,=,=)}S_1$.

A direction vector $\mathbf{d}=(\mathtt{d}_1,\mathtt{d}_2,\ \ldots\ ,\mathtt{d}_n)$ is said to be lexicographically positive if and only if for any $\mathtt{d}_i\mathtt{=>}$ there exist a $\mathtt{d}_j\mathtt{=<}$ where `j<i`.

Even though direction vectors are less accurate in representing dependence relations than distance vectors, they are still frequently used. In fact, in some cases, distance vectors cannot be computed while direction vectors can be. Moreover, some optimizations (e.g. loop permutation) require only direction vectors.

2.3.3 Dependence Cycles

A dependence from a block of statements B_1 to another block of statements B_2 exists, if there exists a dependence $\mathrm{S}_1\delta\mathrm{S}_2$, where S_1 belongs to B_1 and S_2 belongs to B_2.

A dependence cycle, between two blocks of statements B_1 and B_2, indicates the presence of two dependences $\mathrm{B}_1\delta\mathrm{B}_2$ and $\mathrm{B}_2\delta\mathrm{B}_1$. The dependence $\mathrm{B}_1\delta\mathrm{B}_2$ indicates that the block B_2 (i.e. the target) depends on block B_1 (i.e. the source). Similarly, the dependence $\mathrm{B}_2\delta\mathrm{B}_1$ indicates that the block B_1 depends on block B_2.

2.4 Loop and Data Transformations

This section describes loop and data transformations that are frequently used in compilers and are relevant to this research.

```
L1 do i = 1, 10, 1
L3     do k = 1, 10, 1
L2         do j = 1, 10, 1
S1             a(k,j,i) = a(k,j,i) + b(k+1,j-1,i+1) + 7
S2             b(k-1,j+3,i-2) = a(k-2,j+2,i)
E2         enddo
E3     enddo
E1 enddo
```

Figure 2.6: After permuting loops L_2 and L_3 in Figure 2.5, the above nest is obtained. The permutation of loops L_2 and L_3 is legal.

2.4.1 Loop Permutation

Loop permutation interchanges the position of two loops L_i and L_j belonging to a loop nest N [MW96, AK84]. The loop nest N must be a perfect nest. The permutation of L_i and L_j affects the dependence relations in the loop nest. To account for the permutation of L_i and L_j, the elements of all the direction and distance vectors, at the i^{th} and j^{th} positions must be similarly permuted. The permutation is legal if the direction vectors of the dependences in N remain lexicographically positive. Figure 2.6 shows the loop nest of Figure 2.5 after permuting loops L_2 and L_3. After permutation the dependences become the following: $S_1\delta^f_{(2,0,-2)}S_2$, $S_1\delta^a_{(2,3,-4)}S_2$ and $S_1\delta^a_{(0,0,0)}S_1$.

2.4.2 Loop Distribution

Loop distribution, also known as loop fission, splits a loop into multiple loops with the same header but each containing a subset of the statements of the original loop [Kuc77]. Loop distribution is legal if all the statements causing a dependence cycle are kept in the same loop. Figure 2.7 shows an example of loop distribution. There is a dependence cycle between the statements S_1 and S_2 in Figure 2.7(a). Thus these two statements must be kept in the same loop as shown in Figure 2.7(b). In Figure 2.7(c), statements S_1 and S_2 are enclosed by different loops, thus the distribution is not legal.

```
L1 do i = 1, 10, 1
S1     a(i) = a(i) + b(i-1)
S2     b(i) = a(i-1)
S3     c(i) = b(i-2)
E1 enddo
```

(a) original loop

```
L1 do i = 1, 10, 1
S1     a(i) = a(i) + b(i-1)
S2     b(i) = a(i-1)
E1 enddo
L2 do i = 1, 10, 1
S3     c(i) = b(i-2)
E2 enddo
```

(b) legal distribution

```
L1 do i = 1, 10, 1
S1     a(i) = a(i) + b(i-1)
E1 enddo
L2 do i = 1, 10, 1
S2     b(i) = a(i-1)
E2 enddo
L3 do i = 1, 10, 1
S3     c(i) = b(i-2)
E3 enddo
```

(c) illegal distribution

Figure 2.7: Example of loop distribution.

```
L1 do i = 1, 10, 1
B1     body1
E1 enddo
L2 do i = 1, 10, 1
B2     body2
E2 enddo
```

(a) original loops

```
L12 do i = 1, 10, 1
B1      body1
B2      body2
E12 enddo
```

(b) after fusion

Figure 2.8: Example of loop fusion.

2.4.3 Loop Fusion

Loop fusion combines the bodies of multiple loops into one [Kuh80, MA97]. The resulting body is enclosed by a new loop while the original loops are removed. Figure 2.8 shows the fusion of two loops L_1 and L_2 with bodies B_1 and B_2 respectively, into a loop L_{12} whose body is made of B_1 followed by B_2. The fusion is legal if there are no dependences in L_{12} from B_2 to B_1.

Loop fusion can be complicated if the loop headers are not compatible (i.e. same loop bounds and step), or if the loops to be fused are not adjacent (i.e. there is code between them).

```
S BlockOfStatements
L do i = 1, 10, 1
B     body
E enddo
```

(a) original loop

```
L do i = 1, 10, 1
C     if (i == 1) then
S        BlockOfStatements
      endif
B     body
E enddo
```

(b) after sinking

Figure 2.9: Example of code sinking.

2.4.4 Code Sinking

Code sinking moves a block of statements into a loop [MW96]. A guard is used to ensure the sunk code is executed the correct number of times. Figure 2.9 shows the sinking of a block of statements S into a loop L with a body B. The sinking is legal if and only if there are no flow dependences between the block S and the header of the loop L. The *ifstmt* makes sure the block S is executed only once during the first iteration of L.

2.4.5 Strip-Mining

Strip-mining splits a loop into two tightly nested loops [BGS94]. The outer loop, called the controlling loop, steps between strips of consecutive iterations. The inner loop steps between consecutive iterations within a strip. Strip-mining is always legal because it does not change the order of execution of the iterations. Figure 2.10 shows a loop L before and after being strip-mined into two loop L_1 and L_2.

2.4.6 Tiling

Tiling consists of strip-mining two or more tightly nested loops, followed by loop permutation [MW96, WL91b]. The goal of tiling is to change the way the iteration space is traversed, such that the accesses to a region of an array are brought closer in time. Tiling is legal if the loops to be tiled can be permuted. Figure 2.11 shows a nest that implements matrix multiplication before and after tiling.

```
L  do i = 1, n, 1
B      a(i) = b(i) + c(i)
E  enddo
```

(a) original loop

```
L1 do j = 1, n, s
L2     do i = j, min(j+s-1, n), 1
B          a(i) = b(i) + c(i)
E2     enddo
E1 enddo
```

(b) after strip-mining

Figure 2.10: Example of strip-mining.

```
do i = 1, n, 1
  do j = 1, n, 1
    do k = 1, n, 1
      a(k,i)+=b(j,i)*c(k,j)
    enddo
  enddo
enddo
```

(a) original nest

```
do ii = 1, n, s
  do jj = 1, n, s
    do kk = 1, n, s
      do i = ii, min(n,ii+s-1), 1
        do j = jj, min(n,jj+s-1), 1
          do k = kk, min(n,kk+s-1), 1
            a(k,i)+=b(j,i)*c(k,j)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

(b) after tiling

Figure 2.11: Example of tiling.

2.5 Data Reuse and Cache Locality

On the one hand, *data reuse* [MT96, WL91a] occurs when an application accesses the same data multiple times. Data reuse is a function of the application and its data access patterns. On the other hand, cache locality [MT96, WL91a] occurs when the data to be reused is found in the cache. Cache locality depends on the presence of data reuse in the application, the size of the data and the cache parameters (e.g. size and associativity). Data reuse does not always translate into cache locality due to cache line evictions.

2.5.1 Cache Temporal Locality

The cache *temporal* locality indicates that the *same data element* in the cache line is referenced more than once. There are two types of temporal locality.

- *Self-temporal locality* indicates that the same array reference causes the multiple accesses to the same data element. In Figure 2.12, the reference `c(i)` exhibits self-temporal locality.

- *Group-temporal locality* indicates that distinct array references cause the multiple accesses to the same data element. In Figure 2.12, the reference `a(2j,i)` in S_1 and in S_2 exhibit group-temporal locality.

2.5.2 Spatial Locality and Reuse

The cache *spatial* locality indicates that *different data elements* in the same cache line are referenced. The *reuse distance* is the number of data elements between the two referenced elements in the same cache line. Similar to temporal locality, there are two types of spatial locality.

- *Self-spatial locality* indicates that the same array reference causes the accesses to different data elements in the same cache line. In Figure 2.12, the reference `d(j+1,i)` exhibits self-spatial locality.

```
L1 do i = 1, n, 1
L2     do j = 1, n, 1
S1         a(2j,i) = b(4j+1,i) + c(i) + d(2i,j)
S2         d(j+1,i) = a(2j,i) + b(4j,i)
E2     enddo
E1 enddo
```

Figure 2.12: Example of different types of locality.

- *Group-spatial locality* indicates that distinct array references cause the accesses to different data elements in the same cache line. In Figure 2.12, the references `b(4j+1,i)` in S_1 and `b(4j,i)` in S_2 exhibit group-spatial locality.

2.5.3 Data Layout

Data layout refers to the way the elements of a multi-dimensional array are arranged in memory [KRC97, MA95]. Fortran uses column-major storage order for arrays. In the column-major order, two array elements whose index differs by one in the leftmost (first) subscript position are allocated consecutively in memory.

The *stride-1* (also referred to as the fastest changing) dimension of an array is the dimension in which two consecutive elements are mapped onto two consecutive elements on the same cache line.

The data layout of arrays can affect the spatial locality of a reference. The array reference `d(2i,j)`, in Figure 2.12, exhibits no locality. However, by changing the data layout to row-major, the reference would exhibit self-spatial locality.

2.6 Related Work

In his book [MW96], Michael Wolfe discusses techniques that transform imperfect loop nests into perfect ones. However, there is very little research on the metrics and the effectiveness of these techniques. Most proposed algorithms that target perfect nests ignore imperfect nests and do not try to make them perfect.

Carr, McKinley and Tseng [CMT94] use loop distribution to enable loop permutation either by making the nest perfect or by simplifying the dependences. Moreover, they use loop fusion to either increase temporal locality or to make nests perfect. Carr *et al.* report that loop distribution was applied in 12 of the 35 applications considered. Loop distribution was applied to 23 nests and resulted in 29 additional nests. Fusion, however, is found to be unsuccessful in enabling loop permutation. Similarly, this study applies loop distribution and loop fusion to obtain perfect nests. In contrast, this thesis reports, in more detail, on how successful loop distribution and loop fusion are, and on the reasons why they are sometimes ineffective. The study also evaluates scalar expansion that makes loop distribution more effective. Moreover, the thesis measures the execution time overhead introduced by these techniques.

To enhance cache locality, several researchers have proposed a variety of optimizations and algorithms. Carr, McKinley and Tseng [CMT94] study loop permutation to enhance cache spatial locality in sequential programs. A cost model is derived to estimate the number of cache lines accessed when a given loop should be made innermost. A collection of 35 programs and kernels are considered. They report that loop permutation increased the percentage of loop nests, in which the loop with the least cost is innermost, from 74% to 85%. Of the 35 programs, 27 showed degradation or no benefit in performance. Seven programs experienced a speedup of 1% to 13%. One program posted a speedup of 115%. Hence, they conclude that applications are often programmed with good locality. Part of this thesis verifies these results. However, our cost model is simpler, and some run-time profiling is used to attribute different weights to loop nests. Moreover, this study considers the possibility of array layout changes to improve spatial locality.

M. Kandemir, J. Ramajunam and A. Choudhary [KRC97] propose a cache locality enhancing framework that unifies loop and data transformations. The framework considers both loop permutation and changing array layout to improve cache spatial locality. The researchers present results for small kernels chosen from some NAS benchmarks. In contrast, this thesis uses two separate simple frameworks to assess the need for changing the data layout and the need for loop permutation, in 23 complete applications (not only kernels). It should be noted that the unified framework [KRC97] is better than our two separate frameworks, because it considers the interaction between changing the array layouts and loop permutation.

Chapter 3

Methodology

In this chapter, the methodology and the details of the transformations used in this research are presented. First, Section 3.1 deals with some general issues. Then, Section 3.2 highlights the research approach. Sections 3.3, 3.4, 3.5 and 3.6 discuss the details of code sinking, loop distribution, scalar expansion, and loop fusion, respectively. Finally, Section 3.7 deals with spatial locality and presents the loop permutation framework and the data layout framework.

3.1 General Issues

3.1.1 General Definitions

- *If statement* (*ifstmt*) *propagation* refers to moving an *ifstmt* that encloses a loop inside this loop. Consequently, the loop body will consist of the *ifstmt.* The block of statements that used to make up the body of the loop will become the new body of the *ifstmt.* Figure 3.1 illustrates *ifstmt* propagation.

- A *zero-trip* loop is loop which has a trip count equal to zero, i.e. no iterations of the loop are executed.

- We refer to a loop as *safe* if the three following conditions hold:

   1. the loop is not exited nor entered by a *goto* statement,

```
if (cond) then            do i = 1, 10, 1
  do i = 1, 10, 1            if (cond) then
     st                         st1
  enddo                      endif
endif                     enddo
```

(a) N_1 (b) N_1'

Figure 3.1: The nest N_1' is N_1 after *ifstmt* propagation.

2. the loop body has no subroutine calls, and

3. the loop body has no *I/O* statements.

3.1.2 Benchmarks Used

In total, 23 applications are considered. Table 3.1 lists all the applications and their respective benchmark suites. Moreover, the table shows the abbreviations used in the results chapters (Chapters 4 and 5). The applications are chosen from 4 suites: Perfect Club [BCP+89], NAS [BBB+91], SPEC92 [Dix92] and NCSA [NCS]. Some of the figures presented in this thesis include an entry labelled AV. This entry represents the average for all the applications shown.

3.2 Research Approach

The first goal of this research is to investigate how the relative contribution to execution time of perfect nests can be increased by using the three transformations, namely, code sinking, loop distribution, and loop fusion. Figure 3.2 shows a block diagram of the system used.

The block labelled *Polaris + New Passes* represent the Polaris [BEF+95] compiler and the passes added for this research. Polaris is a compiler for "Automatic Parallelization of Conventional Fortran Programs", from the University of Illinois [BDE+96]. Polaris is a source-to-source restructuring compiler, whose main objective is to detect parallel

Benchmark Suites	**Applications**	**Abbreviations**
NAS	appbt	BT
	embar	EP
	fftpde	FT
	buk	IS
	applu	LU
	mgrid	MG
	appsp	SP
PerfectClub	qcd	LG
	mdg	LW
	track	MT
	bdna	NA
	ocean	OC
	dyfesm	SD
	arc2d	SR
	flo52q	TF
	trfd	TI
Spec92	hydro2d	HY
	ora	OR
	su2cor	SU
	swm256	SW
	tomcatv	TO
	wave5	WV
NCSA	cmhog	HG

Table 3.1: Applications examined and their respective benchmark suites.

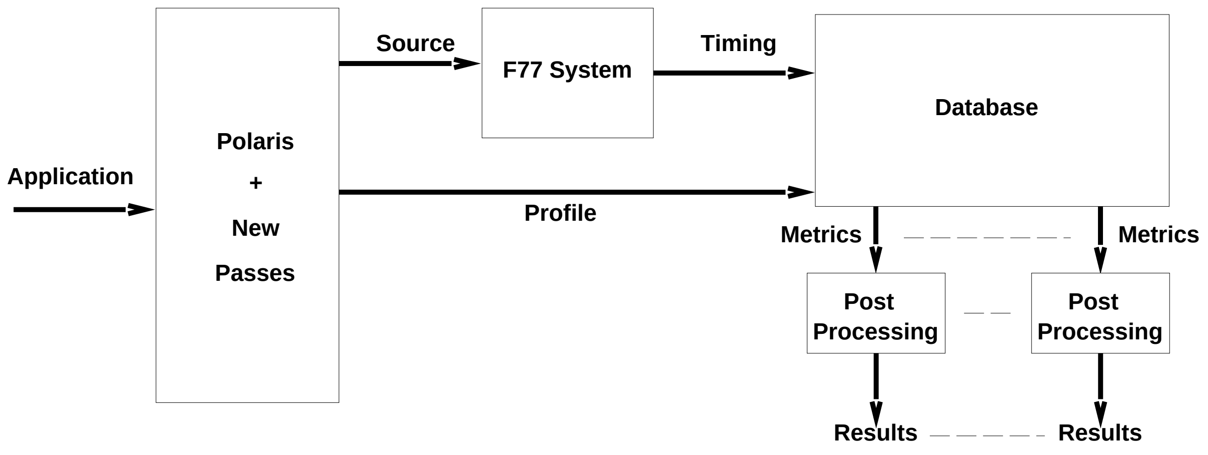


Figure 3.2: Block diagram of the approach used

loops. It consists of multiple passes (e.g. constant propagation, dead code elimination) that are applied in sequence. The block labelled *F77 System* represents a system that compiles and executes Fortran 77 programs. The *Database* represents the storage for the information provided by Polaris and the F77 System. Finally, the *Post Processing* units query the database and produce specific results.

The application source is processed, using Polaris, to generate an instrumented source and a set of profile information. The source is instrumented to measure execution time of loop nests and subroutine calls in the program. The profile contains information such as the number of loops and the nesting structures. The instrumented code is then compiled using a native F77 compiler, and the executable is used to generate timing information. This timing information along with the profile information is stored in a database. This database allows subsequent queries that derive the various metrics presented in this thesis, such as the relative contribution to execution time of nests that may benefit from tiling.

The main advantage of this approach stems from storing timing and profile information for later reuse. The time it takes to run Polaris and execute some of the applications is 1000 times greater than the time it takes to do the post processing. Hence, the information in the database can be accessed multiple times without the need to re-run Polaris or to re-examine the original source code. Thus, the approach makes getting results very efficient and fast. Moreover, the approach is extensible; to get some new results, simply a new post processing module needs to be added.

```
do i = $i_b$, $i_f$, 1
    st1
    do j = $j_b$, $j_f$, 1
        st2
    enddo
    st3
enddo
```

(a) N_1

```
do i = $i_b$, $i_f$, 1
    do j = $j_b$, max($j_b$,$j_f$), 1
        if (j == $j_b$) then
          st1
        endif
        if ($j_b \leq j_f$) then
          st2
        endif
        if (j == max($j_b$,$j_f$)) then
          st3
        endif
    enddo
enddo
```

(b) N_1'

```
do i = $i_b$, $i_f$, 1
    do j = $j_b$, $j_f$, 1
        if (j == $j_b$) then
          st1
        endif
        st2
        if (j == $j_f$) then
          st3
        endif
    enddo
enddo
```

(c) N_1''

Figure 3.3: The nest N_1 is imperfect. The nest N_1' is N_1 made perfect by code sinking. N_1'' is N_1 made perfect by code sinking when the inner loop is known to execute at least once.

3.3 Code Sinking

3.3.1 Mechanics of Code Sinking

Code sinking can be used to transform an imperfect nest into a perfect one [MW96]. The program segments in Figure 3.3 illustrate this transformation. The nest N_1, in Figure 3.3(a), is imperfect because of the statements *st1* and *st3*. Code sinking pushes theses two statements inside the inner loop, as shown in Figure 3.3(b). Guards around the pushed statements are needed to ensure that they are executed the correct number of times. Since it may not be known whether the inner loop is a zero-trip loop, its bounds must be changed to have a trip count of at least 1. Moreover, the original statements of the inner loops must be guarded to execute only if the original inner loop executes at least once. However, if it is known that $j_b \leq j_f$ then the bounds of the inner loop need not be changed and the guard around the statement *st2*— i.e. the inner loop statements— would not be required, as shown in Figure 3.3(c).

Code sinking is only legal if dependences allow it. For example, in Figure 3.3(a), a flow dependence between *st1* and the header of the inner loop would make code sinking illegal. Also, in this research we do not apply code sinking in loops that are not safe, because of the complications involved. For example, the label of a goto statement that exits the loop, may need to be redefined, especially when there is code after the loop that has be sunk.

3.3.2 Inner Loops Enclosed by *ifstmts*

One code structure that code sinking, as well as loop distribution and loop fusion, must deal with is a nest where loops are enclosed by *ifstmts*. This structure is dealt with here and can also be handled similarly for loop distribution and for loop fusion. Figure 3.4 shows an example of a nest N_2, where the inner loop is enclosed by an *ifstmt*, and how it is handled to give a perfect nest N_2'. The statement *st1* is sunk in the inner loop to give an *ifstmt* that tightly[1] encloses the inner loop. Then, the *ifstmt* is propagated inside the inner loop. Thus, the *ifstmt* becomes the only statement in the inner loop. Furthermore, the statements that used to make up the body of the inner loop constitute the body of the *ifstmt*. The following assumptions are made:

- The conditional expression *c1* is assumed to have no side effects. Otherwise, it would be illegal to sink it into the inner loop, because it will be evaluated $itrip*jtrip$ times instead of *itrip* times, where *itrip* and *jtrip* are the trip counts for the outer and inner loops, respectively.
- There are no flow dependences from statements *st1* and *st2* to the condition *c1*.
- The inner loop of N_2 is known to be a non zero-trip loop. Otherwise, the bounds of the inner loop must be changed and guards must be placed around *st2*.

The technique described above is not applied directly when the *ifstmt* encloses more then one loop at the same nesting level. Instead, the inner loops must first be fused. Then the *ifstmt* is propagated inside the inner loop. Another option would be to propagate the *ifsmt* into each loop and then fuse the loops. This case is mostly relevant when trying

[1]The *ifstmt* has only one statement, which is the inner loop.

```
do i = imin, imax, 1
   if (c1)
     st1
     do j = jmin, jmax, 1
        st2
     enddo
   endif
enddo
```

(a) N_2

```
do i = imin, imax, 1
   do j = jmin, jmax, 1
      if (c1)
        if (j == jmin)
          st1
        endif
        st2
      endif
   enddo
enddo
```

(b) N'_2

Figure 3.4: The nest N_2 is imperfect and has an if statement that encloses a loop. The nest N'_2 is N_2 made perfect by code sinking and *ifstmt* propagation.

to apply loop fusion to obtain perfect nests, and it will be discussed in more details in Section 3.6.

3.3.3 Nests with Loops at Same Nesting Level

Figure 3.5 shows how sinking a loop inside another loop can transform a nest N_3, where two inner loops are nested at the same level, into a perfect nest N'_3. The statement *st1* can be sunk in the first inner loop. However, this will not make N_3 perfect because the outer loop has two inner loops nested at the same level. An option is to sink the first inner loop in the second inner loop. This will make it possible to obtain a perfect nest with a depth of 3. In the general case, if there were n inner loops at the same nesting level, then code sinking the inner loops will result in a perfect nest with depth *n+1*. However, in this research, we do not implement this technique, because we believe it will lead to very complicated code structures, especially in nests with depth greater than 2, and with more than two loops nested at the same level. Furthermore, loop distribution and loop fusion are used to handle nests exhibiting this structure.

```
do i = imin, imax, 1
    st1
    do j = jmin, jmax, 1
        st2
    enddo
    do k = kmin, kmax, 1
        st3
    enddo
enddo
```

(a) N_3

```
do i = imin, imax, 1
    do k = kmin, kmax, 1
        do j = jmin, jmax, 1
            if (k == kmin)
                if (j == jmin)
                    st1
                endif
                st2
            endif
            if (j == jmax)
                st3
            endif
        enddo
    enddo
enddo
```

(b) N_3'

Figure 3.5: The nest N_3 has inner loops nested at the same level. Using code sinking to push loops inside other loops, the nest N_3 can be transformed into a perfect nest N_3'.

```
$L_1$ do i = 1, 10, 1
$S_1$     a(i) = b(i) + c(i)
$L_2$     do j = 1, 10, 1
$S_2$         d(i,j) = b(j) + c(j)
$E_2$     enddo
$E_1$ enddo
```

(a) N_1

```
$L'_1$ do i = 1, 10, 1
$S'_1$     a(i) = b(i) + c(i)
$E'_1$ enddo
$L''_1$ do i = 1, 10, 1
$L''_2$     do j = 1, 10, 1
$S''_2$         d(i,j) = b(j) + c(j)
$E''_2$     enddo
$E''_1$ enddo
```

(b) N'_1 and N''_1

Figure 3.6: The nest N_1 is imperfect. The nests N'_1 and N''_1, are the results of applying loop distribution to N_1.

3.4 Loop Distribution

3.4.1 Mechanics of Loop Distribution

Loop distribution can be used to transform an imperfect nest into a perfect one, as shown in Figure 3.6. The original imperfect nest N_1, is split into two distinct nests, N'_1 and N''_1. The statement S_2, common to both loops L_1 and L_2 in N_1, becomes the body S''_2 of the perfect nest (L''_1,L''_2) in N''_1. The statement S_1, enclosed by the outer loop L_1 only, constitutes the body S'_1 of the loop L'_1 in N'_1.

3.4.2 Legality of Loop Distribution

The legality of loop distribution is determined by the dependences between the blocks of statements to be placed in different loops. Loop distribution is legal if and only if there exists no dependence cycles, as defined in Section 2.3.3, between the blocks [Kuc77]. Dependences carried by outer loops that enclose the loop to be distributed do not affect the legality of distribution because they will always be satisfied by the outer loops[MW96].

Figure 3.7 shows an example where the legality of loop distribution depends on the blocks of statements to be split. The relevant dependences are $S_2\delta^f_{=}S_3$ due to the references to the array `a`, and $S_3\delta^f_{<}S_2$ due to the references to the array `c`. On the one hand, Figure 3.7(b) shows the distribution when statements S_2 and S_3 are considered

```
$L_1$ do i = 1, 10, 1
$S_1$     d(i) = a(i)
$S_2$     a(i) = c(i)
$S_3$     c(i+1) = a(i)
$E_1$ enddo
```

(a) Original loop

```
$L_1'$ do i = 1, 10, 1
$S_1'$     d(i) = a(i)
$E_1'$ enddo
$L_1''$ do i = 1, 10, 1
$S_2''$     a(i) = c(i)
$S_3''$     c(i+1) = a(i)
$E_1''$ enddo
```

(b) Legal

```
$L_1'$ do i = 1, 10, 1
$S_1'$     d(i) = a(i)
$S_2'$     a(i) = c(i)
$E_1'$ enddo
$L_1''$ do i = 1, 10, 1
$S_3''$     c(i+1) = a(i)
$E_1''$ enddo
```

(c) Illegal

Figure 3.7: Loop distribution, as in (c) is illegal because the distributed blocks, caused by $S_2\delta^f_= S_3$ and $S_3\delta^f_< S_2$. Loop distribution in (b) is legal because the above dependences are in the same block.

a block, and the statement S_1 is considered another. The distribution is legal because there are no dependence cycles between the two blocks. On the other hand, Figure 3.7(c) shows the distribution when statements S_1 and S_2 are considered a block B_{12}, and the statement S_3 is considered another block B_3. The distribution is illegal because there is a dependence cycle between B_{12} and B_3. The cycle is made of $B_{12}\delta^f_= B_3$ caused by $S_2\delta^f_= S_3$, and of $B_3\delta^f_< B_{12}$ caused by $S_3\delta^f_< S_2$.

3.4.3 Scalar Expansion

The definition and use of a scalar variable in a loop may give rise to loop-carried anti and output dependences [MW96]. Loop distribution may become illegal due to a *scalar dependence cycle*. A scalar dependence cycle is a dependence cycle that will cease to be a cycle once scalar *anti* and *output* dependences are removed.

Scalar expansion[PW86] is a transformation that replaces references to a scalar in a loop by references to an array. Each loop iteration references a different element of the array. Consequently, scalar expansion replaces scalar *flow* dependences with loop-independent dependences. More importantly, scalar expansion eliminates the scalar *anti* and *output* dependences. Hence, scalar expansion may enable loop distribution by removing such dependences and thus breaking up scalar dependence cycles.

Figure 3.8 illustrates the scalar expansion transformation and how it can enable loop

```
$L_1$ do i = $i_b$, $i_f$, 1
$S_1$     r = a(i)
$L_2$     do j = $j_b$, $j_f$, 1
$S_2$         b(i,j) = r
$E_2$     enddo
$E_1$ enddo
```

(a) Original

```
$L_1$ do i = $i_b$, $i_f$, 1
$S_1$     rx(i) = a(i)
$L_2$     do j = $j_b$, $j_f$, 1
$S_2$         b(i,j) = rx(i)
$E_2$     enddo
$E_1$ enddo
```

(b) Scalar expansion

```
$L_1'$ do i = $i_b$, $i_f$, 1
$S_1'$     rx(i) = a(i)
$E_1'$ enddo
$L_1''$ do i = $i_b$, $i_f$, 1
$L_1''$     do j = $j_b$, $j_f$, 1
$S_2''$         b(i,j) = rx(i)
$E_2''$     enddo
$E_1''$ enddo
```

(c) Loop distribution

Figure 3.8: The nest in (a) cannot be made perfect by loop distribution because there exists a scalar dependence cycle caused by the variable `r`. After expanding `r` into a one dimensional array `rx` in (b), loop distribution is applied to yield (c)

distribution. In the program segment shown in Figure 3.8(a), L_1 is the loop to be ditributed. The blocks to be distributed are: S_1 constitutes the first block B_1 while L_2, S_2 and E_2 constitute the second block B_2. Loop distribution is illegal because there is a dependence cycle between B_1 and B_2. The cycle consists of $S_1\delta^f_= S_2$, which flows from B_1 to B_2, and $S_2\delta^a_< S_1$ which flows from B_2 to B_1. Figure 3.8(b) shows the result after scalar expansion of the scalar `r`. The scalar anti-dependence $S_2\delta^a_< S_1$ is removed and the dependence cycle between B_1 and B_2 is broken. Therefore, loop distribution becomes legal, as shown in Figure 3.8(c). The mechanics and legality of scalar expansion are discussed in detail in Section 3.5.

3.4.4 Reasons for the Ineffectiveness of Loop Distribution in Obtaining Perfect Nests

There are several reasons that disallow loop distribution from obtaining perfect nests. They are classified below:

- UNSAFE: the loop considered is not a *safe* loop. If the loop is entered or exited by a *goto* statement then applying loop distribution becomes very complicated. The presence of I/O or subroutine calls requires inter-procedural analysis which is not available for this research. Thus, in this case, loop distribution is considered unable to transform an imperfect nest into a perfect one.

- STRUCTURE: the structure of the nest will disallow distribution. The two blocks to be placed in different loops might not be separable. In fact, even after trying *ifstmt* propagation, there may remain an *ifstmt* whose header and tail statements are in the two different blocks to be distributed.

- ARRAYDEP: there exists a dependence cycle that prevents distribution. This cycle is made up of only array dependences and scalar flow dependences. In this case, scalar expansion would not enable distribution.

- SCALARDEP: all the dependence cycles that prevent distribution are scalar dependence cycles, as defined in Section 3.4.3. In this case, scalar expansion can enable distribution.

In applying loop distribution, the reason for failure in obtaining a perfect nest is noted. If a loop is found to be *unsafe* then the reason for distribution failure is marked as UNSAFE, and the other conditions are not checked. Analogously, if SCALARDEP is marked as the reason for failure then the loop is *safe*; its structure allows distribution; there are no dependence cycles made of only array dependences; and finally there exists at least one scalar dependence cycle.

3.5 Scalar Expansion

Scalar expansion is a transformation that replaces a scalar variable, in a loop, by an array variable [PW86, MW96]. Each element of the new array variable replaces the scalar variable for one iteration of the loop. Scalar expansion eliminates *anti* and *output* dependences caused by references to the scalar. Also, *flow* dependences will be replaced by loop-independent dependences. Hence, scalar expansion can be used to eliminate scalar anti and output dependences that are part of a dependence cycle, allowing loop distribution to be applied.

```
      REAL r
      do i = 1, 10, 1
S1       r = ...
S2       ... = r
      enddo
```

(a) N

```
      REAL r, rx(10)
      do i = 1, 10, 1
S1'      rx(i) = ...
S2'      ... = rx(i)
      enddo
```

(b) N′

Figure 3.9: The loop dependent dependence, in the nest N, is eliminated by expanding the scalar r into an array rx (nest N′).

3.5.1 Mechanics of Scalar Expansion

Figure 3.9 illustrates the application of scalar expansion. The references to the scalar `r` are replaced by references to the array `rx`. The number of elements of `rx` should be at least equal to the trip count of the loop considered. In the nest N, there are two dependence relations. First, there is the flow dependence $S_1 \delta^f_= S_2$, which becomes $S'_1 \delta^f_= S'_2$ in the nest N′. Second, there is the anti dependence $S_1 \delta^a_< S_2$, which is eliminated in N′. Thus, scalar expansion eliminated the loop carried anti dependence.

There are some factors that complicate the application of scalar expansion. First, if the scalar to be expanded is alive after the loop then the correct value of the scalar must be copied from the appropriate array element to the scalar after the loop is exited. Figure 3.10 illustrates this case and shows how it is handled. Second, if there is an upward-exposed use[2] of the scalar in the loop, then the flow dependence is a loop carried dependence. Hence, the use and the definition of the scalar in the loop would be replaced by references to consecutive array elements. Figure 3.11 illustrates this case and shows how correctness is preserved. Third, if scalar expansion is to be applied for a loop nest with a dimension greater than 1, then the number of elements in the array should be equal to the multiplication of all the trip counts of the loops in the considered nest. Hence, the size of the array may be very large which may cause considerable execution

[2]An upward-exposed use of a variable A means that the variable A is used before it is defined inside the loop. During the first iteration of the loop, the value of the variable to be used comes from outside the loop.

```
REAL r
do i = 1, 10, 1
    r = ...
    ... = r
enddo
... = r
```

(a) N

```
REAL r, rx(10)
do i = 1, 10, 1
    rx(i) = ...
    ... = rx(i)
enddo
r = rx(10)
... = r
```

(b) N′

Figure 3.10: The scalar r is used after the loop. Thus, it must be assigned the correct value at the exit of the loop.

```
REAL r
r = ...
do i = 1, 10, 1
    ... = r
    r = ...
enddo
```

(a) N

```
REAL r, rx(10+1)
r = ...
rx(1) = r
do i = 1, 10, 1
    ... = rx(i)
    rx(i+1) = ...
enddo
```

(b) N′

Figure 3.11: The use of r in the nest N is upward-exposed. The nest N′ shows how to apply scalar expansion to r. It should be noted that the flow dependence between rx(i+1) and rx(i) is a loop carried dependence.

```
REAL r
do i = 1, 10, 1
    do j = 1, 5, 1
        do k = 1, 15, 1
            r = ...
            ... = r
        enddo
    enddo
enddo
```

(a) N

```
REAL r, rx(15,5,10)
do i = 1, 10, 1
    do j = 1, 5, 1
        do k = 1, 15, 1
            rx(k,j,i) = ...
            ... = rx(k,j,i)
        enddo
    enddo
enddo
```

(b) N′

Figure 3.12: To eliminate the loop carried anti-dependence, the scalar r must be expanded such that there is a distinct array element for every triplet (k,j,i).

time overhead. When loop bounds are compile-time constants, a multi-dimensional array can be used to replace the scalar. Each dimension of the array would correspond to a loop of the nest. Figure 3.12 illustrates this case and how it is handled. The use of multi-dimensional arrays makes indexing straightforward. Finally, if the bounds of the loop are not compile-time constants then static arrays cannot be used. Instead dynamically-allocated arrays must be used. Figure 3.13 illustrates this case and how it can be handled for a loop nest with dimension equal to 1. The syntax to express the dynamic allocation, use and deallocation of arrays in this figure, conforms to that of the SGI f77 compiler. The following remarks clarify some of the constructs used in the code segments in Figure 3.13.

- The declaration *POINTER (rxptr, rx)* states that the pointer *rxptr* and the array *rx* are associated. The pointer *rxptr* is used to allocate the array. The array *rx* is used to access the array elements.
- The declaration *DIMENSION rx(0:0)* indicates that *rx* is a uni-dimensional array where the number of elements is not specified.
- The statement *rxptr = malloc(imax*8)* allocates *imax*8* bytes. These bytes will be pointed to by *rxptr*. A variable of type *REAL* is assumed to be 8 bytes long.

```
REAL r
do i = 1, imax, 1
    r = ...
    ... = r
enddo
```

(a) N

```
REAL r, rx, rxptr
POINTER (rxptr, rx)
DIMENSION rx(0:0)
rxptr = malloc(imax*8)
do i = 1, imax, 1
    rx(i-1) = ...
    ... = rx(i-1)
enddo
CALL free(rxptr)
```

(b) N′

Figure 3.13: The upper bound, imax, of the loop in the nest N is not a compile-time constant. To apply scalar expansion correctly, the array, used replace the scalar, must be declared dynamically, as shown in nest N′.

- The array *rx* is indexed from *0* to *imax-1*. This explains the use of the array reference *rx(i-1)*.
- The statement *CALL free(rxptr)* deallocates the memory used for the array *rx*.

Figure 3.14 illustrates the case of variable loop bounds and a 2-dimnesional loop nest. In this case, the number of array elements allocated is equal to the multiplication of the trip counts of all the loops in the nest. Moreover, indexing the array that replaces the scalar is complicated because the 2-dimensional iteration space is mapped onto a one dimensional array.

3.6 Loop Fusion

3.6.1 Mechanics of Loop Fusion

Loop fusion can be used to transform an imperfect nest into a perfect one [CMT94]. Figure 3.15 illustrates this transformation. The nest N_1 has two inner loops L_2 and L_3 at the same nesting level. The perfect nest N_1' is obtained after fusing these two inner loops. The headers of the loops L_2 and L_3 need not be compatible. The lower bound of the resulting loop L_{23}' is equal to the minimum of the lower bounds of the loops to

```
REAL r
do i = 1, imax, 1
   do j = 1, jmax, 1
      r = ...
      ... = r
   enddo
enddo
```

(a) N

```
REAL r, rx, rxptr
POINTER (rxptr, rx)
DIMENSION rx(0:0)
rxptr = malloc(imax*jmax*8)
do i = 1, imax, 1
   do j = 1, jmax, 1
      rx((j-1) + jmax*(i-1)) = ...
      ... = rx((j-1) + jmax*(i-1))
   enddo
enddo
CALL free(rxptr)
```

(b) N′

Figure 3.14: In nest N, scalar expansion is to be applied in two nested loops whose upper bounds are variable. The nest N′, shows the result after applying scalar expansion to the nest N. It should be noted that the use of a two-dimensional array is not possible, because its dimensions are not known constant. Thus, a one dimensional array must be used. Hence, the complex array subscripts.

be fused. Similarly, the upper bound of the resulting loop L'_{23} is equal to the maximum of the upper bounds of the loops to be fused. The guards C'_1 and C'_2 ensure that the bodies of L_2 and L_3 respectively, are executed the correct number of times. Also, the loop indices are unified and the array subexpressions are modified accordingly.

When applying loop fusion to obtain a perfect nest, the loops to be fused may be enclosed by a common *ifstmt*, similar to nest N$_2$ shown in Figure 3.16. To transform a nest such N$_2$ into a perfect one, the loops are fused inside the *ifstmt*. Then the *ifstmt* is propagated in the fused loop, to give a perfect nest N$'_2$ in Figure 3.16. This example also shows the case when the loops to be fused have compatible headers. Another way to handle this case is to propagate the *ifstmt* into the the two loops and then fuse the loops. However, we do not adopt this strategy because it adds more code to the body of the loops, hence decreasing the chance of having a legal fusion.

```
L_1 do i = i_b, i_f, 1
L_2     do j = j_b, j_f, 1
S_2         st1(j)
E_2     enddo
L_3     do k = k_b, k_f, 1
S_2         st2(k)
E_3     enddo
E_1 enddo
```

(a) N_1

```
L'_1  do i = i_b, i_f, 1
L'_23     do h = min(j_b,k_b), max(j_f,k_f), 1
C'_1          if (j_b ≤ h ≤ j_f)
S'_1              st1(h)
              endif
C'_2          if (k_b ≤ h ≤ k_f)
S'_2              st2(h)
              endif
E'_2      enddo
E'_1  enddo
```

(b) N'_1

Figure 3.15: The nest N_1 has inner loops nested at the same level. Fusing the two inner loops, transforms the nest N_1 into a perfect nest N'_1.

3.6.2 Legality of Loop Fusion

Fusing two adjacent loops L_1 and L_2 with bodies B_1 and B_2 respectively, to yield L_{12} with a body made of B_1 followed by B_2, is legal if the following conditions are satisfied. First, in L_{12} there should be no dependences, unless carried by an outer loop, whose source is B_2 and sink is B_1 [MW96, MA97]. Second, the header of each loop to be fused should not depend on the body of the other loop. The loop headers need not be compatible as illustrated in Figure 3.15.

Figure 3.17 illustrates a case where loop fusion is illegal and a case where it is legal. Fusing L_2 and L_3 to give L_{23} as shown in Figure 3.17(b) is illegal. In fact, in L_{23}, there is a dependence $S_3\delta^a_< S_2$ caused by the refernces to the array c. This dependence indicates that in every iteration, S_3 uses an element of c that will be written by S_2 in the next iteration. The original nest in Figure 3.17(a), however, indicates that the elements of c read by S_3 will be previously defined by S_2, which is no longer the case in the nest of Figure 3.17(b).

In Figure 3.17(c), the fusion of L_1 and L_2 into L_{12} is legal because in L_{12} there is no dependence whose source is S_2 and sink is S_1. The only dependence in L_{12} is $S_1\delta^f_= S_2$ caused by the references to the array a.

```
do i = i_b, i_f, 1
   if (c1) then
      do j = j_b, j_f, 1
      st2
      enddo
      do j = j_b, j_f, 1
         st3
      enddo
   endif
enddo
```

(a) N_2

```
do i = i_b, i_f, 1
   do j = j_b, j_f, 1
      if (c1)
         st1
         st2
      endif
   enddo
enddo
```

(b) N'_2

Figure 3.16: The nest N_2 has inner loops nested at the same level. These inner loops are also enclosed by a common *ifstmt*. Using loop fusion and *ifstmt* propagation, N_2 is transformed into a perfect nest N'_2.

```
L1 do i = 1, 10, 1
S1      a(i) = b(i) + 2
E1 enddo
L2 do i = 1, 10, 1
S2      c(i) = a(i) * 2
E2 enddo
L3 do i = 1, 10, 1
S3      d(i) = c(i+1)
E3 enddo
```

(a) Original nest

```
L1 do i = 1, 10, 1
S1      a(i) = b(i) + 2
E1 enddo
L23 do i = 1, 10, 1
S2      c(i) = a(i) * 2
S3      d(i) = c(i+1)
E23 enddo
```

(b) Illegal fusion

```
L12 do i = 1, 10, 1
S1      a(i) = b(i) + 2
S2      c(i) = a(i) * 2
E12 enddo
L3 do i = 1, 10, 1
S3      d(i) = c(i+1)
E3 enddo
```

(c) Legal fusion

Figure 3.17: Fusion of L_2 and L_3, as shown in (b), is illegal (see text). However, fusion of $L_1$and L_2, as shown in (c) is legal.

3.6.3 Reasons for the Ineffectiveness of Loop Fusion in Obtaining Perfect Nests

There are several reasons that disallow loop fusion from obtaining perfect nests. They are classified below.

- UNSAFE: one of the loops to be fused is an *unsafe* loop, i.e. it is entered or exited by a *goto* statement. The presence of I/O or subroutine calls requires inter-procedural analysis which is not available for this research. Thus, in this case, loop fusion is considered unable to transform an imperfect nest into a perfect one.
- STRUCTURE: the structure of one of the nests disallows fusion. For instance, even after trying *ifstmt* propagation, there may remain an *ifstmt* that encloses one of the loops to be fused. Therefore, the *ifstmt* prevents the loops from being adjacent.
- INTERCODE: some code between the loops to be fused cannot be moved to make the loops adjacent.
- BACKDEP: a dependence in the fused loop indicates that fusion is illegal, as seen in Section 3.6.2.
- HEADERDEP: there is a dependence between the body of one of the loops to be fused and the header of the other.

In applying loop fusion to two loops L_2 and L_3 that are enclosed by L_1, the above categories, are examined, in the listed order. For instance, if one of the three loops is found to be *unsafe* then the reason for fusion failure is marked as UNSAFE, and the other conditions are not checked. Analogously, if HEADERDEP is marked as the reason for failure, then all three loops are *safe*; their structure allows fusion; there is no intercode that cannot be moved between the loops to be fused L_2 and L_3; there are no backward dependences in the resulting fused loop; and finally there exists a dependence between the header of L_2 and the body of L_3 or vice-versa.

3.6.4 Fusion Algorithm

Our fusion algorithm targets a loop that directly encloses multiple loops with possible straight-line code (called INTERCODE) between them. It tries to fuse directly enclosed loops and sink straight-line code in order to obtain a perfect nest. There are more than one variations of handling the straight-line code between the loops to be fused. In fact, the order of applying code sinking to handle the INTERCODE and fusing adjacent loops may affect the legality of loop fusion.

Indeed, it is beneficial to always fuse adjacent loops before sinking code in any of the loops. Figure 3.18 illustrates why. There are two ways of obtaining N_{final} from N_{org}. First, S_1 can be sunk in L_2 and then L_2 and L_3 can be fused. In N_{cs} which is obtained from N_{org} after code sinking, the fusion of L_2 and L_3 is illegal because there would be backward dependence in the resulting loop from S_3 to S_1. Thus, for N_{org}, code sinking followed by loop fusion is illegal. However, L_2 and L_3 can be fused, which is legal, then S_1 is sunk in the resulting loop, which is also legal. Thus, for N_{org}, loop fusion followed by code sinking is legal. Ideally, sinking followed by fusion and fusion followed by sinking are equivalent because they yield the same transformed code. However, most compilers are conservative when dealing with control dependences and thus fusion after code sinking may be illegal. Code sunk inside a loop is executed only once either at the first iteration or the last iteration of the loop. Most compilers assume that the sunk code may be executed at every iteration of the loop, hence the additional dependences that make fusion illegal.

The algorithm in Figure 3.19 implements loop fusion in order to obtain perfect nests. The strategy of the algorithm is to delay code sinking as much as possible. Code sinking is applied only when there are no more adjacent loops.

First, the algorithm starts by making a copy of the whole nest (i.e. L_{outer}). A copy is needed to restore the nest to its original state, in case the algorithm is not successful in obtaining a perfect nest.

Second, all the adjacent loops are fused. When two adjacent loops cannot be fused then the whole algorithm fails and the original nest must be restored. If the number of loops directly enclosed by L_{outer}, becomes equal to one, then all the directly enclosed

```
L1 do i = 1, 10, 1
S1     a(i) = b(i) + 2
L2     do j = 1, 10, 1
S2         c(i,j) = c(j,i)
E2     enddo
L3     do j = 1, 10, 1
S3         d(i,j) = a(j)
E3     enddo
E1 enddo
```

(a) Original nest N_{org}

```
L1 do i = 1, 10, 1
L2     do j = 1, 10, 1
C1         if (j == 1) then
S1             a(i) = b(i) + 2
C2         endif
S2         c(i,j) = c(j,i)
E2     enddo
L3     do j = 1, 10, 1
S3         d(i,j) = a(j)
E3     enddo
E1 enddo
```

(b) After code sinking, N_{cs}

```
L1 do i = 1, 10, 1
S1     a(i) = b(i) + 2
L2     do j = 1, 10, 1
S2         c(i,j) = c(j,i)
S3         d(i,j) = a(j)
E2     enddo
E1 enddo
```

(c) After fusion, N_{fuse}

```
L1 do i = 1, 10, 1
L2     do j = 1, 10, 1
C1         if (j == 1) then
S1             a(i) = b(i) + 2
C2         endif
S2         c(i,j) = c(j,i)
S3         d(i,j) = a(j)
E2     enddo
E1 enddo
```

(d) Final nest N_{final}

Figure 3.18: Example of a code segment where applying code sinking before loop fusion might make loop fusion illegal. The nest N_{org} is the original nest. The nest N_{cs} is N_{org} after sinking S_1 in L_2. For N_{cs}, fusion of L_2 and L_3 is not legal. The nest N_{fuse} is N_{org} after fusing L_2 and L_3. Finally, the nest N_{final} is obtained from N_{fuse} by sinking S_1 in L_2.

```
Input:  A loop $L_{outer}$
Output: A boolean indicating if a perfect nest is obtained
        $L_{outer}$ is modified to obtain the perfect nest

begin
  //Make a copy of the original loop to be restored if fusion is not successful
  Loop Copy = clone($L_{outer}$)
  SetofLoops DirectlyEnclosed = GetDirectlyEnclosedLoops($L_{outer}$)
  //if $L_{outer}$ directly encloses 0 or 1 loops then fusion
  //is not applicable. Code sinking should be used if there is only 1 enclosed
  //loop. If there are no enclosed loops then nothing needs to be done.
  if(DirectlyEnclosed.Cardinal() < 2)
     return FALSE;
  endif
  while(TRUE)
    while(DirectlyEnclosed has two loops $L_{in_1}$ and $L_{in2}$ that are adjacent)
       $L_{fused}$ = fuse($L_{in_1}$, $L_{in2}$)
       if(Fusion failed)
          $L_{outer}$ = Copy  //restore the original $L_{outer}$
          return FALSE
       endif
       DirectlyEnclosed.Remove($L_{in_1}$)
       DirectlyEnclosed.Remove($L_{in2}$)
       DirectlyEnclosed.Add($L_{fused}$)
    endwhile

    //check if all the inner loops were fused into one
    if(DirectlyEnclosed.Cardinal() == 1)
       ApplyCodeSinking($L_{outer}$)
       if(Sink succeeded)
          return TRUE
       endif
       else
          $L_{outer}$ = Copy  //restore the original $L_{outer}$
          return FALSE
       endelse
    endif
    else  //there are at least two loops $L_{in_1}$ $and$ $L_{in2}$
          //in DirectlyEnclosed with code between them
       SinkIntercode($L_{in_1}$,$L_{in2}$)
       if(Sink failed)
          $L_{outer}$ = Copy  //restore the original $L_{outer}$
          return FALSE
       endif
    endelse
  endwhile
end
```

Figure 3.19: Algorithm Used in Applying Loop Fusion to Obtain Perfect Nests

loops by L_{outer} are fused into one loop. Thus, sinking the code between L_{outer} and the remaining inner loop would make the nest perfect. In this case, fusion succeeds in transforming the imperfect nest into a perfect one.

However, if there are no more adjacent loops to fuse and the number of loops directly enclosed by L_{outer}, is greater than one, then intercode between two loops must be moved to make the loops adjacent. If code sinking fails in making the two loops adjacent then the algorithm fails. Otherwise, there are two new adjacent loops, and the algorithm goes back to the stage of fusing the adjacent loops.

Handling the intercode between two loop requires more explanation. The intercode handler (`SinkInterCode()`) tries to make the two loops L_{in1} and L_{in2} adjacent. It determines the block of statements B_{12} that needs to be sunk. Then it computes the dependences between B_{12} and L_{in1} on the one hand, and between L_{in2} and B_{12} on the other. These dependences determine where the block of statement B_{12} can or must be sunk. If there are no dependences between B_{12} and neither loop, then B_{12} can be sunk in either loop. However, if there are dependences between B_{12} and both loops, then code sinking in any of the loops is useless because loop fusion of L_{in1} and L_{in2} would later fail. If there are dependences between B_{12} and only one of the loops, then B_{12} must be sunk in the loop with which it has a dependence. If B_{12} is sunk in the loop with which it has no dependence then the fusion of L_{in1} and L_{in2} would later fail.

3.7 Spatial Locality

This section deals with locality in general and spatial locality in particular. First, a characterization of array references is presented. Second, the framework used to assess the need for data layout to enhance spatial locality is discussed. Third, the framework used to assess the need for loop permutation to enhance spatial locality is presented. This research does not present a framework to combine both loop permutation and array layout changes. We make two assumptions in our work on locality. First, arrays are assumed to be stored in Fortran's column-major order. This assumption is not restrictive and does not affect the generality of the work. Second, multiple (i.e. more than one) data elements are assumed to fit in a cache line; otherwise, no spatial locality can occur.

3.7.1 Array Reference Characterization

This section characterizes individual array references in terms of what transformations are needed to achieve spatial locality for each reference. In order to have spatial locality, certain conditions, regarding array subscript expressions and the order of loops in a nest, must exist. The conditions for self-spatial locality are different than those for group-spatial locality.

3.7.1.1 Self-Spatial Locality

Self-spatial locality depends on the subscript expressions of the array reference and on the loops ordering of its enclosing nest [KRC97, MT96, WL91a]. Given an array reference, the following conditions must exist to have self-spatial locality.

- One of the loop index variables appears only in the stride-1 dimension of the array reference.
- The coefficient of the index variable mentioned above, should be either 1 or -1.
- The position of the corresponding loop must be innermost in the nest.

Given an array reference, it can be classifed into one of the following categories. These categories reflect how the given array reference can be affected to exhibit self-spatial locality, if possible.

- **Reference not enclosed:** the array reference is not enclosed by any loop. Hence, self-spatial locality is not possible.

- **Inner temporal:** the array reference exhibits self-temporal locality with respect to the inner loop (see Section 2.5.1). Thus, spatial locality may only be obtained, if possible, by permuting the innermost loop with one of the outer loops, which in this case will disturb the temporal locality. Therefore, we believe that, in this case, obtaining spatial locality would not be beneficial. Figure 3.20 illustrates this case. For a given value of the index `i`, all the iterations of loop `j` access the same element of array `A`. Therefore, the array reference `A(i)` exhibit self-temporal locality with respect to the inner loop `j`. Moreover, the reference `A(i)` exhibit self-spatial

```
do i = 1, 10, 1
   do j = 1, 10, 1
      ... = A(i) + ...
   enddo
enddo
```

Figure 3.20: Example of self-temporal locality with respect to the innermost loop

reuse with respect to the outer loop `i`. To increase the potential of translating this reuse into spatial locality, the `i` loop should be permuted to the innermost position. However, by doing so, the loop carrying the self-temporal locality will be made outermost, which may diminish the temporal locality.

- **Spatial locality not possible:** the subscript expressions of the array reference prevent spatial locality. The reference `a[3i,4j]` is an example of this case. Indeed, the reference `a[3i,4j]` cannot exhibit self-spatial locality because both of the indices, in the subscript expressions, have coefficients that are different than 1 and -1.

- **Already in spatial order:** the array reference exhibits self-spatial locality already.

- **Loop Permutation Needed:** even though spatial reuse may be present due to accesses to the same cache line, it may not be translated into spatial locality if the loop carrying the locality is not innermost. The reason for this is that the multiple accesses to the same cache line are separated by the iterations of inner loops. Therefore, the accessed cache line may be evicted before it is reused. By making the loop that carries the reuse innermost, the accesses to the same cache line would become closer in time which increases the chance of obtaining spatial locality. Figure 3.21 illustrates this case. Two consecutive iterations of the `i` loop, access two consecutive elements of array `a` in the same cache line. However, since the `i` loop is outermost, consecutive iterations of this loop are separated by `jmax-jmin` iterations of the `j` loop. Thus, the cache line may be evicted before the next iteration of the `i` loop occurs. By making the `i` loop innermost, there will

```
do i = imin, imax, 1
    do j = jmin, jmax, 1
        sum = sum + A(i,j)
    enddo
enddo
```

(a) N_1

```
do j = jmin, jmax, 1
    do i = imin, imax, 1
        sum = sum + A(i,j)
    enddo
enddo
```

(b) N_1'

Figure 3.21: The nest N_1 has no spatial locality. The nest N_1', obtained from N_1 by loop permutation, has spatial locality from the array reference A(i,j).

be no `j` loop iterations between consecutive iterations of the `i` loop. Thus, spatial locality may be improved.

- **Layout Change Needed:** in some array access patterns, consecutive elements in an array dimension D may be accessed. If dimension D is the stride-1 dimension than the accessed elements will be in the same cache line and thus there will be spatial locality. However, if dimension D is not the stride-1 dimension then the accessed elements will be in different cache lines and there would be no spatial locality. By changing the array layout such that dimension D becomes the stride-1 dimension would cause spatial locality. Figure 3.22 illustrates this case. By storing the array A in row-major order, the subscript expression of the stride-1 dimension becomes `j` instead of `2i`. And since the `j` loop is innermost, spatial locality will result.

- **Layout Change and Loop Permutation Needed:** this case is the combination of the two cases above. Changing the array layout is needed to obtain spatial reuse, and loop permutation is needed to translate this reuse into locality. Figure 3.23 illustrate this case. Two consecutive iterations of the `i` loop access two consecutive elements in a row of array `A`. However, since array `A` is stored such that a cache line contains consecutive elements of a column of `A`, there is no spatial reuse. By laying out `A` in row-major order, the two consecutive iterations of the `i` loop would access two elements of `A` in the same cache line. Even though at this stage there exist spatial reuse, the accesses to the same cache line are separated by `jmax-jmin`

```
do i = imin, imax, 1
    do j = jmin, jmax, 1
        sum = sum + A(2i,j)
    enddo
enddo
```

Figure 3.22: The array reference A(2i,j) exhibits no spatial locality because it is stored in column-major order. If array A is stored in row-major order, then the index j will be in the stride-1 dimension, and spatial locality will be achieved.

```
do i = imin, imax, 1
    do j = jmin, jmax, 1
        sum = sum + A(2j,i)
    enddo
enddo
```

(a) N_1

```
do j = jmin, jmax, 1
    do i = imin, imax, 1
        sum = sum + A(2j,i)
    enddo
enddo
```

(b) N_1'

Figure 3.23: The nest N_1 has no spatial locality. The nest N_1' is obtained from N_1 by loop permutation. And if array A is stored in row-major order then spatial locality will be achieved.

iterations of the `j` loop, during which the cache line may be evicted. By permuting the `i` and `j` loops, the two accesses to the same cache line will be in two consecutive iterations of the inner loop `i`. Thus spatial locality would be obtained.

- **Permutation or layout needed:** either loop permutation or a change in the array layout is needed and sufficient to get self-spatial locality for the array reference.

3.7.1.2 Group-Spatial Locality

Group-spatial locality depends on the subscript expressions of the array references involved. Given an array reference, it can be classifed into one of the following categories. These categories reflect how the given array reference can be affected, to exhibit group-spatial locality, if possible.

- **Temporal group:** the reference belongs to a group of references exhibiting group-temporal locality. Two array references A_1 and A_2, to the same array, are considered to exhibit group-temporal locality with respect to an enclosing loop L if either of the next two conditions hold [CMT94]:

1. there exists a loop-independent dependence caused by the references to A_1 and A_2, or
2. there exists a loop-carried dependence $S_1 \delta S_2$ represented by distance vector $\mathbf{d}$ = $(d_1, \ldots, d_N)$, and caused by the references to A_1 and A_2, where $\|d_L\| \leq 2$, and all other entries in $\mathbf{d}$ are zero.

In this case, changing the array layout is considered not beneficial because the reference already exhibits group-temporal locality.

- **Spatial group:** the reference belongs to a group of references exhibiting group-spatial locality. Two array references A_1 and A_2, to the same array, are considered to exhibit group-spatial locality if the two following conditions hold [CMT94]:

  1. All array subscripts in A_1 and A_2 must be pairwise identical except the subscripts in the stride-1 dimension.
  2. The absolute value of the difference of the subscripts in the stride-1 dimension of A_1 and A_2 must be greater than zero and smaller than the cache line size in array elements.

- **Layout change needed:** the above two conditions listed for "Spatial group" may hold for a dimension D that is different than the stride-1 dimension. If the layout of the array can be changed such that dimension D becomes the stride-1 dimension, then group-spatial locality may be obtained. Figure 3.24 illustrates this case. The two references `A(3j,4i)` and `A(3j,4i+1)` access two elements of A that are consecutive and in the same row. If A is stored in column-major order then cache lines would contain consecutive array elements of the same column. Thus, `A(3j,4i)` and `A(3j,4i+1)` will exhibit no group-spatial locality. However, if A is stored in row-major order than the elements accessed by the two references would be in the same cache line. Thus group-spatial locality would be exhibited. Loop

```
do i = 1, 10, 1
    do j = 1, 10, 1
        B(j,i) = A(3j,4i) + A(3j,4i+1)
    enddo
enddo
```

Figure 3.24: By laying out the array A in row-major order, then the two array reference to A would exhibit group spatial locality.

permutation does not affect group-spatial locality because it does not affect the stride-1 dimension of arrays.

3.7.2 Data Layout Framework

This section describes the framework used to assess whether changing the data layout of arrays is needed to enhance spatial locality. First, it should be pointed out that this framework does not check for the legality of changing the data layout. Data layout may be illegal due to array aliasing. Second, this framework only considers changing the array layout as an intra-procedural transformation as opposed to a global transformation. Hence, the effects of changing the layout of an array, in a given procedure, are not considered on other procedures. Because of the above two limitations, the framework presented here will yield optimistic results as to the benefit of layout change. In other words, when the legality and the inter-procedural effects of changing array layouts are taken into accout, the benefit of data layout would be smaller than the one reported by this framework.

This framework considers conflicts between references to the same array. In fact, two array references A_1 and A_2 to the same array A, may require different layouts to exhibit spatial locality. By choosing an array layout to make A_1 exhibit spatial locality then A_2 would not exhibit spatial locality, and vice-versa. It should be noted that the layout of a given array is considered static, i.e. it remains the same during the whole program.

To estimate the effect of allowing different layouts for different arrays, the algorithm in Figure 3.25 is applied to the benchmark applications. First, the algorithm divides array references into groups. Each group contains all the references to a given array. Second,

```
Input:  Pgm: A program
Output: $N_{orig}$: Number of array references exhibiting self
               or group-spatial locality without any change in layout
        $N_{max}$: Maximum number of array references exhibiting self
               or group-spatial locality  when the array layout can be
               different for different arrays

begin
  $N_{orig}$ = 0
  $N_{max}$ = 0

  //group the arrays such that all references to the same array are together
  SetofArrayGroups arraygroups = Pgm.GroupArrays()

  //iterate over all the groups of arrays
  for(Iterator groupiter = arraygroups; groupiter.valid(); ++groupiter)
    //iterate over the dimensions of the array that represents the
    //current arraygroup
    for(Iterator dimiter = groupiter.current().array().dimensions();
                               dimiter.valid(); ++dimiter)
      //count and saves the number of references with self or group-spatial
      //locality when the current dimension is the fastest changing
      groupiter.current().CountSpatialRefs(dimiter.current())
    endfor

    //get the number of references with spatial locality for the
    //dimension originally fastest changing, and add it to $N_{orig}$
    $N_{orig}$ += groupiter.current().GetOrgCount()

    //get the maximum number of references with spatial locality for
    //any dimension made fastest changing, and add it to $N_{max}$
    $N_{max}$ += groupiter.current().GetMaxCount()
  endfor
end
```

Figure 3.25: Algorithm used to assess the effects of data layout on the array references with self or group-spatial locality.

for each group, each dimension of the array is considered as the stride-1 dimension, and the number of references exhibiting self or group-spatial locality is recorded. The total number ($\mathtt{N}_{orig}$) of array references with spatial locality for the original stride-1 dimension for all the arrays is computed. The maximum number $\mathtt{N}_{max}$ of array references with spatial locality for any dimension in the stride-1 position for all the arrays is also computed. Hence, N_{max} is the maximum total number of array references, in the application, that exhibit group or self-spatial locality, when having different layouts for different arrays is allowed. If $\mathtt{N}_{max}$ is found to be greater than $\mathtt{N}_{orig}$, then flexible layout is considered desirable.

3.7.3 Loop Permutation Framework

This section describes the framework used to assess whether loop permutation is needed to enhance spatial locality. This framework considers conflicts between array references in the same nest. In fact, two array references may require two different loops in the innermost position of the enclosing nest.

For every nest, an index is calculated for every loop when it is placed innermost. Then, it is decided whether loop permutation is desirable. The decision is made by comparing the maximum index calculated with the index of the original inner loop. If the maximum index is larger than the index of the original inner loop, then loop permutation is considered desirable. The index is simply the count of array references that exhibit spatial locality when the considered loop is innermost. More precisely, for each nest N with loops (L_1, L_2, ... , L_n), a vector (M_1, M_2, ... , M_n) is calculated, where M_i is the number of array references exhibiting spatial locality when loop L_i is made innermost. M_n is the number of array references with spatial locality for the loop L_n which is originally innermost. M_{max} is the maximum of the elements in the vector (M_1, M_2, ... , M_n), and corresponds to the highest number of array references exhibiting spatial locality if any loop in the nest N can be made innermost. To decide if loop permutation is desirable M_{max} is compared to M_n. If ($\mathrm{M}_{max} > \mathrm{M}_n$) then loop permutation is considered desirable, otherwise it is not.

Chapter 4

Improvements in Nest Structures

This chapter reports on the experimental results concerning loop nests structures. Section 4.1 describes how loops and nests are timed. Section 4.2 describes the nest structures before any transformation. Sections 4.3, 4.4 and 4.5 respectively present the effects on nest structures and the overhead of code sinking, loop distribution with and without scalar expansion, and loop fusion. Finally, Section 4.6 compares the three techniques and draws general conclusions.

4.1 Timing Technique

Since this research uses the relative contribution to execution time of loop nests as a metric, it is important to explain how the timing of loops is conducted. First, the following terms are defined.

1. Total Time
   - The *total time* of a loop is the time it takes to execute the loop.
   - The *total time* of a perfect nest is the *total time* of its outer loop.
2. Net Time
   - The *net time* of a loop is equal to its *total time* minus the *total time* of all the loops and subroutines directly enclosed by it.
   - The *net time* of a perfect nest is equal to the *net time* of the innermost loop of the perfect nest.

```
beg = time()
do i = $i_b$, $i_f$, 1
    ⋮
enddo
end = time()
looptime += end - beg
```

(a) Loop timing

```
beg1 = time()
do i = $i_b$, $i_f$, 1
    ⋮
    beg2 = time()
    do j = $j_b$, $j_f$), 1
        ⋮
    enddo
    end2 = time()
    looptime2 += end2 - beg2
enddo
end1= time()
looptime1 += end1 - beg1
```

(b) Loop nest timing

Figure 4.1: Loop and loop nest timing

To time a given loop L, calls to timing routines are inserted before the header and after the tail of the loop. The execution time of L is equal to the difference between the two timing calls. However, the loop L may be executed multiple times[1]. Thus, the execution time of L needs to be accumulated. Figure 4.1(a) illustrates how a loop is timed. Figure 4.1(b) shows how loops in nests are timed.

It should be noted that the insertion of timing calls is intrusive. That is, the timing calls affect the execution time of the application. However, we believe that the effects of timing routines are small.

4.1.1 Timing of Nests Made Perfect by Code Sinking

This subsection describes how the execution time for nests made perfect by code sinking is derived. Given two loops L_{outer} and L_{inner} that can be made tightly nested using code sinking as Figure 4.2 shows, then the times of the resulting perfect nest and the involved loops are computed as follows. The total time of L'_{outer} is considered equal to the time of L_{outer}, i.e. the overhead of the guards used in code sinking is ignored. The overhead is

[1]The loop L may be inside a subroutine that is called multiple times, or the loop L may be enclosed by an outer loop.

```
$L_{outer}$ do i = $i_b$, $i_f$, 1
            Code X
$L_{inner}$     do j = $j_b$, $j_f$, 1
              ⋮
            enddo
        enddo
```

(a) Original

```
$L'_{outer}$ do i = $i_b$, $i_f$, 1
$L'_{inner}$     do j = $j_b$, $j_f$, 1
                  Code X
                  ⋮
             enddo
         enddo
```

(b) Code sinking

Figure 4.2: Loop times after code sinking.

ignored in order to keep execution time of perfect nests from being artificially inflated. The total time of L'_{inner} is given by:

$$L'_{inner}.\text{total} = L_{inner}.\text{total} + L_{outer}.\text{net}$$

The equation tries to approximate the addition of the execution time of the sunk code to the total time of L_{inner}. It is only an approximation because the net time of L_{outer} includes the execution time for the header and tail of L_{outer}. We believe that this is a reasonable approximation because usually the body of a loop contribute to most of the execution time of the loop. The net times of the loops and the new perfect nest are recomputed as described in the previous section.

4.1.2 Timing of Nests Made Perfect by Loop Distribution

This subsection describes how the execution time for nests made perfect by loop distribution is derived. Given a loop L_{outer} enclosing a loop L_{inner}, and given that L_{outer} can be distributed to obtain a single loop L_{single} and a perfect nest (L'_{outer},L'_{inner}), as shown in Figure 4.3, then the times of the resulting loops are computed as follows. The total time of L'_{inner} is equal to the total time of L_{inner}. The total time of L'_{outer} is approximated as the total time of L_{inner}. The header and tail times of L'_{outer} are not included. They are expected to be small compared to the total time of L_{inner}. The total time of L_{single} is given by:

$$L_{single}.\text{total} = L_{outer}.\text{total - } L_{inner}.\text{total}$$

```
                                 L_single  do i = i_b, i_f, 1
                                              Code X
L_outer  do i = i_b, i_f, 1                 enddo
            Code X               L'_outer  do i = i_b, i_f, 1
L_inner     do j = j_b, j_f, 1   L'_inner     do j = j_b, j_f, 1
               ⋮                                 ⋮
            enddo                             enddo
         enddo                             enddo

      (a) Original                    (b) Loop Distribution
```

Figure 4.3: Loop times after loop distribution.

The equation states that the total time of the new loop L_{single} is equal to the execution time of the code between L_{outer} and L_{inner} plus the time of the header and tail of L_{outer}. The net times of the loops and the new perfect nest are recomputed as described in Section 4.1.

4.1.3 Timing of Nests Made Perfect by Loop Fusion

This subsection describes how the execution time for nests made perfect by loop fusion is derived. Given a loop L_{outer} enclosing loops L_{in1} and L_{in2}, and given that these inner loops can be fused to obtain a loop L_{inner} which will be part of a perfect nest (L_{outer},L_{inner}), as Figure 4.4 shows, then the times of the resulting loops are computed as follows. The total time of L_{outer} is unchanged. The total time of L_{inner} is approximated as the addition of the total times of L_{in1} and L_{in2}. This is an approximation because it ignores the saving in time due to combining the headers of L_{in1} and L_{in2}. The net times of both loops and the perfect nest are recomputed as described in Section 4.1.

4.1.4 Times of Pseudo and Real Perfect Nests

As mentioned before, real perfect nests are also pseudo perfect nests by definition. Hence, when computing the times for these types of nests, some clarifications are required.

Figure 4.5(a) shows two perfect nests. The nest N_1 = (L_1,L_2) is pseudo perfect, while

```
L_outer do i = i_b, i_f, 1
L_in1      do j = j_b, j_f, 1
               Body of L_in1
           enddo
L_in2      do j = j_b, j_f, 1
               Body of L_in2
           enddo
        enddo
```

(a) Original

```
L_outer do i = i_b, i_f, 1
L_inner    do j = j_b, j_f, 1
               Body of L_in1
               Body of L_in2
           enddo
        enddo
```

(b) Loop Fusion

Figure 4.4: Loop times after loop fusion.

the nest N_3 = (L_3,L_4) is real perfect. On the one hand, when computing the time of real perfect nests, only the time of N_3 is included. The time of N_1 is not included because N_1 is not a real perfect nest. On the other hand, when computing the time for pseudo perfect nests, the time of N_1 is included. The time for N_3 is not considered even though it is also a pseudo perfect nest; the time of N_3 has already been accounted for by the time of N_1.

Figure 4.5(b) shows the real perfect nest N_{13}, obtained by applying code sinking to N_1 and N_3. N_{13} is a real perfect nest. In this case, the times for pseudo and real perfect nests are identical, and are equal to the time of N_1. The time of real perfect nests increases while the time of pseudo perfect nests remains the same.

Figure 4.6 illustrates a different case where the time for pseudo perfect nests increases while the time for real perfect nests remains the same. In Figure 4.6(a) there are no perfect nests. After code sinking, and assuming that the code cannot be sunk further down into L_3, Figure 4.6(b) has one pseudo perfect nest (L_1,L_2) and still no real perfect nests. Hence, the time of pseudo perfect nests will increase while the time for real perfect nests remains the same.

```
L1 do i = ib, if, 1
L2     do j = jb, jf, 1
           ⋮
L3         do k = kb, kf, 1
L4             do m = mb, mf, 1
                   ⋮
               enddo
           enddo
       enddo
   enddo
```

(a) N_1 and N_3

```
L1 do i = ib, if, 1
L2     do j = jb, jf, 1
L3         do k = kb, kf, 1
L4             do m = mb, mf, 1
                   ⋮
                   ⋮
               enddo
           enddo
       enddo
   enddo
```

(b) N_{13}

Figure 4.5: Example to illustrate how the time of real perfect nests may increase while the time of pseudo perfect nests remains the same.

```
L1 do i = ib, if, 1
       S1
L2     do j = jb, jf, 1
           S2
L3         do k = kb, kf, 1
               S3
           enddo
       enddo
   enddo
```

(a) Original

```
L1 do i = ib, if, 1
L2     do j = jb, jf, 1
           S1
           S2
L3         do k = kb, kf, 1
               S3
           enddo
       enddo
   enddo
```

(b) After Code Sinking

Figure 4.6: Example to illustrate how the time of pseudo perfect nests may increase while the time of real perfect nests remains the same.

4.2 Characterization of Execution Time

This section characterizes the execution time of the benchmark applications. It starts with a coarse-grain characterization, according to which execution time is spent in loops or outside loops. This characterization examines the assumption that most execution time is spent in loops. Then, a fine-grain characterization of execution time is presented. The fine-grain characterization considers execution time spent in different nest structures.

4.2.1 Coarse-Grain Characterization

To gain a better understanding of the characteristics of the applications considered, the execution time of an application is characterized as follows.

1. The execution time of code enclosed by a loop.

2. The execution time of code not enclosed by a loop.

The above characterization is important because most optimizations target loops. In fact, it is assumed that the relative contribution to execution time of the loops, in a given data parallel application, is high (more than 90%).

Table 4.1 is derived using the system described in Section 3.2. It summarizes the measured relative contribution to execution time of loops in the benchmark applications. In particular, table 4.1 shows the following:

- For 13 applications, the relative contribution to execution time of loops is greater than 98%. Such high relative contribution to execution time of loops is expected and indeed desired.

- For 19 applications, the relative contribution to execution time of loops is greater than 90%.

- The applications LG, LW, MT, OR and WV have significantly lower relative contribution to execution time of loops. The high relative contribution to execution time of unenclosed code is due to a large number of calls to subroutines and functions that contain no loops. However, when these subroutines and functions are

%Loop Time	**%UnEnclosed Time**	**App**
99.81	0.19	BT
99.95	0.05	EP
94.82	5.18	FT
98.06	1.94	IS
99.93	0.07	LU
99.79	0.21	MG
99.87	0.13	SP
51.17	48.83	LG
91.43	8.57	LGI
75.51	24.49	LW
99.99	0.01	LWI
53.66	46.34	MT
92.87	7.13	MTI
95.63	4.37	NA
98.85	1.15	OC
95.39	4.61	SD
99.04	0.96	SR
99.98	0.02	TF
99.98	0.02	TI
98.41	1.59	HY
20.27	79.73	OR
99.99	0.01	ORI
96.16	3.84	SU
99.92	0.08	SW
99.90	0.10	TO
77.89	22.11	WV
92.54	7.46	WVI
99.97	0.03	HG

Table 4.1: Coarse-grain characterization of execution time. LGI, LWI, MTI, ORI and WVI are partially-inlined versions (see text) of LG, LW, MT, OR and WV respectively.

inlined[2], all five applications, indicated by LGI, LWI, MTI, ORI and WVI, exhibit relative contribution to execution time of loops higher than 90%. Hence, the inlined versions of these applications are used subsequently.

4.2.2 Fine-Grain Characterization

As mentioned above, a high relative contribution to execution time of loops is desirable because most optimizations target the loops in an application. However, some optimizations are designed to target specific loop structures. For example, neither tiling nor permutation can be applied to a loop nest of dimension equal to 1. The following is a refined characterization of execution time.

- **Unenclosed time:** the time for code segments that are not enclosed by any loop.
- **Single time** The net time of single loops.
- **Perfect time:** the net time of real perfect nests.
- **Code-motion time:** the net time of nests that may be made real perfect using code sinking. Nests that are characterized under this category can also be made real perfect by loop distribution. Loop fusion, however, cannot transform such nests into perfect ones.
- **Fusion time:** the net time of nests that require loop fusion to become real perfect. Nests that are characterized under this category can also be made real perfect by loop distribution. Code sinking, however, does not target such nests[3].

Figure 4.7 shows the execution time characterizations of the 23 benchmarks. In the applications SR, HY and SW, perfect nests contribute to almost 100% of the execution time. Therefore, none of the transformations considered in this research are needed for these applications. In MG and TF, perfect nests contribute to a high percentage of execution time. Thus the effects of the transformations is expected to be minimal. The

[2]Only the routines which have no loops and that are called a large number of times are inlined. Inlining some of the applications fully is not practical because of compilation times involved.

[3]If sinking loops into other loops is allowed then code sinking would target these nests. However, we do not consider this case as indicated in Section 3.3.3.

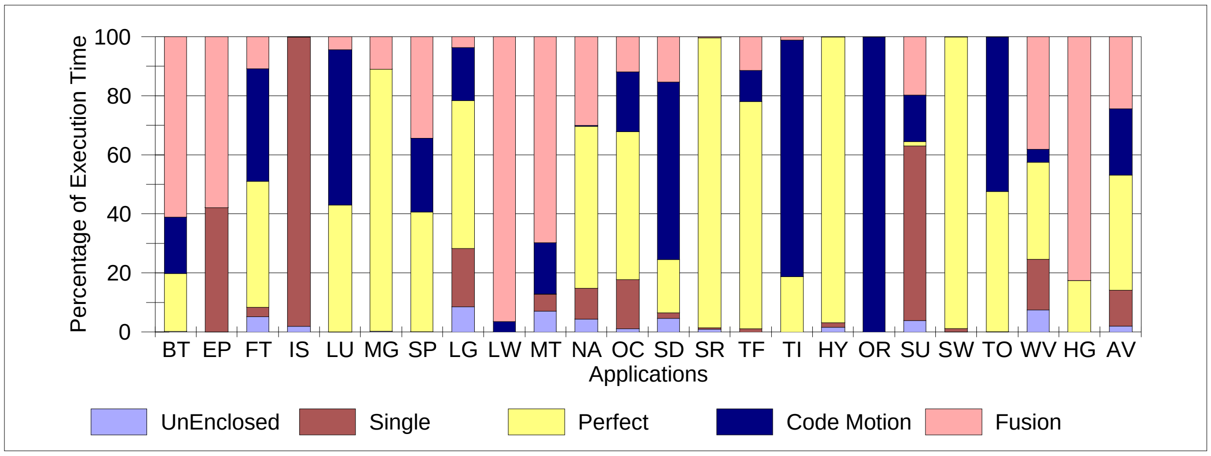


Figure 4.7: Characterization of execution time of the applications.

execution time labelled as "Unenclosed" and "Single" is not targetable by any of the transformations. Therefore the relative contribution to execution time of perfect nests in IS and SU cannot be increased. Code sinking targets the percentage of execution time labelled as "Code Motion". Therefore, it has the potential to significantly improve the contribution to execution time of perfect nests in FT, LU, SP, OC, TI, OR and TI. Loop distribution targets the percentage of execution time classified under the categories "Code Motion" and "Fusion". Loop distribution is expected to target the applications with significant execution time in these two categories. Finally, loop fusion only handles nests whose time is classified as "Fusion". Thus, loop fusion can potentially increase the relative contribution to execution time of perfect nests in BT, EP, SP, LW, MT, NA, WV and HG. From the above, it can be conclude that the three transformations may be needed to increase the contribution to execution time of real perfect nests.

4.3 The Effect of Code Sinking on Nest Structures

This section reports on how code sinking affects the contribution of perfect nests in the applications. It also discusses the overhead introduced by code sinking.

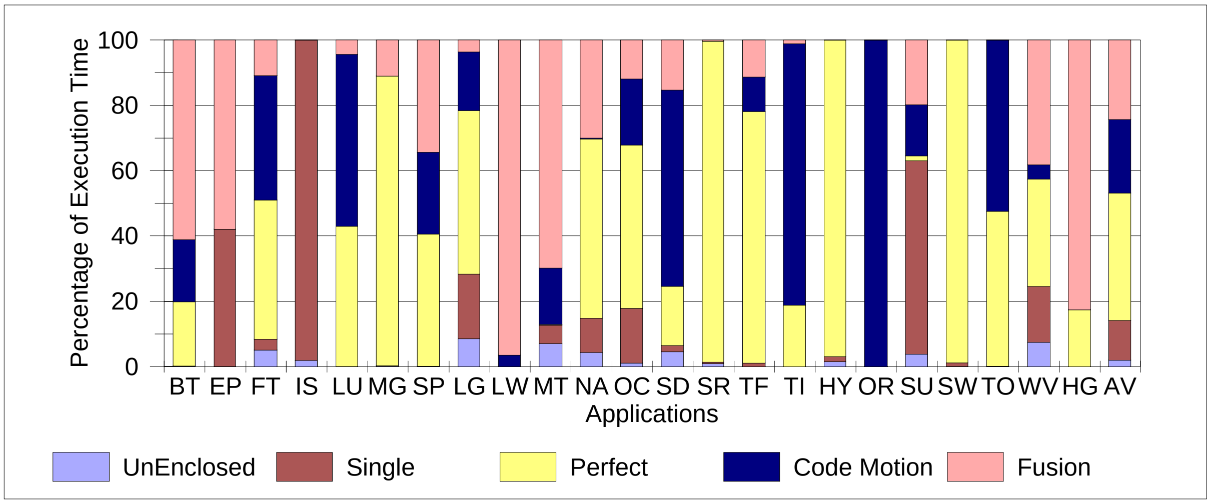
Figure 4.8(a) and Figure 4.8(b) show the characterization of execution time before and after code sinking, respectively. The figures indicate that code sinking is effective in transforming most nests labelled as "Code Motion" into real perfect nests. In Figure 4.8(b), there remain little execution time labelled as "Code Motion". However, code motion is not successful in transforming some imperfect nests into perfect nests due to unsafe loops and dependences. For instance, in OR, there is an unsafe loop that contribute to most of the execution time. In general, code sinking is successful in increasing the relative contribution to execution time of perfect nests in the applications that have a significant portion of execution time classified as "Code Motion". The applications that have a low percentage of execution time labelled as "Code Motion" are not affected by code sinking.

Code sinking increases the relative contribution to execution time of real perfect nests in 13 benchmarks. Figure 4.9 shows the increase in the relative contribution to execution time of pseudo and real perfect nests, for these applications.

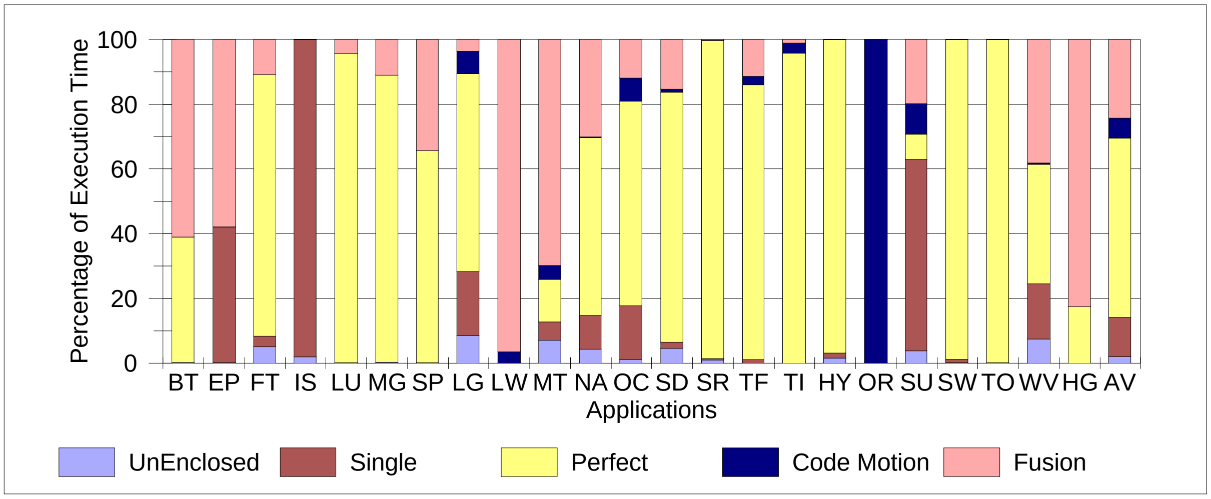
By examining Figure 4.9, it can be seen that, for BT, LU, SP and LG, code sinking increases only the relative contribution to execution time of real perfect nests, while it has little or no effect on the pseudo perfect nests contribution. There are two reasons for this. First, there is a number of pseudo perfect nests that are transformed into real perfect nests and since real perfect nests are also pseudo perfect nests, the relative contribution to execution time of real perfect nests will increase, while the one for pseudo perfect nests remains unchanged (see Section 4.1.4). Second, if a nest that is made real perfect by code sinking, is already enclosed by a pseudo perfect nest, then the execution time of the new perfect nest will not be added to the time of pseudo perfect nests; otherwise the time of the new perfect nest would be accounted for twice.

The computational overhead, due to code sinking, stems from the execution of the guards used to ensure proper execution. Although the sunk code will be executed only once for the trip count of the inner loop, the *ifstmt* used to guard the sunk code will be executed trip count times. The addition of the *ifstmt* to the inner loop increases execution time because the arithmetic operations to evaluate the conditional require additional cycles. Moreover, the conditional may cause a pipeline control hazard or a cache miss.

To assess the computational overhead due to code sinking, the execution times of the

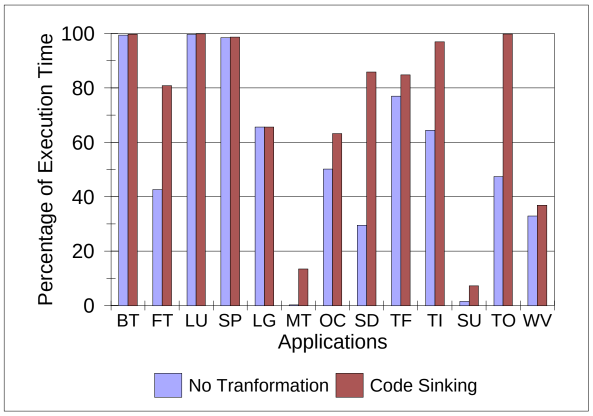


(a) Original

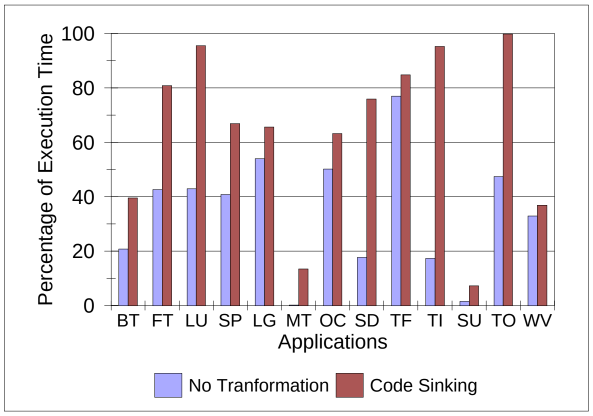


(b) After Code Sinking

Figure 4.8: Characterization of execution time before and after code sinking.



(a) Pseudo Perfect Nests



(b) Real Perfect Nests

Figure 4.9: Contribution to execution time of pseudo and real perfect nests before and after code sinking.

applications after code sinking, are compared with the execution times of the original applications. Figure 4.10, shows the overhead of code sinking for the applications affected by this transformation. Code sinking caused no overhead in 8 of the 13 applications in which it increased the relative contribution to execution time of perfect nests. For 3 other applications, the overhead is less than 5%. Thus, for most applications code sinking introduces acceptable overhead. However, code sinking may be prohibitive for SD and TI because of the high overhead of 35% and 28%, respectively. The large overhead in these two applications is due to sinking code in loops that have small bodies (one or two statements). Consequently, the guards used to ensure proper execution have an execution time that is comparable to that of the body of the original loop. Therefore, special attention should be paid when sinking code into loops with small bodies.

4.4 The Effect of Loop Distribution on Nest Structures

This section reports on how loop distribution with and without scalar expansion affects the number of perfect nests and their relative contribution to execution time in the applications. It also discusses the overhead introduced by loop distribution.

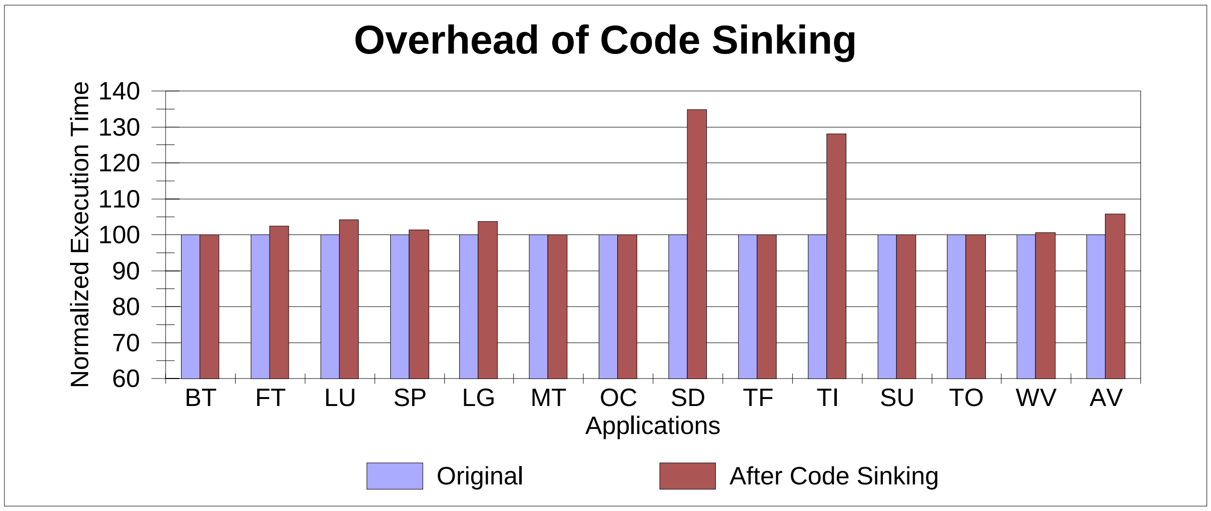


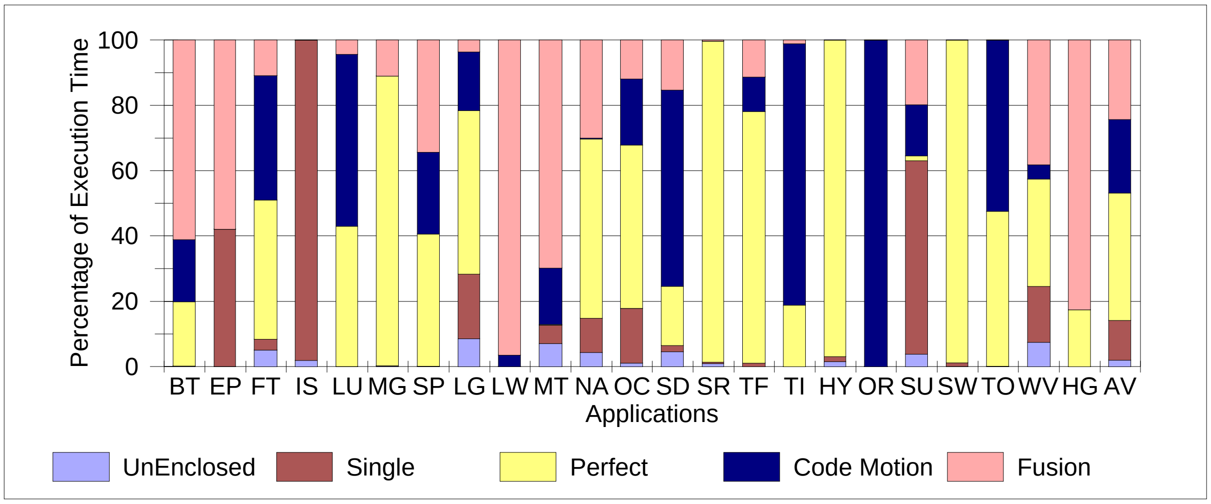
Figure 4.10: Overhead of code sinking on the applications that benefited from this transformation.

4.4.1 Loop Distribution without Scalar Expansion

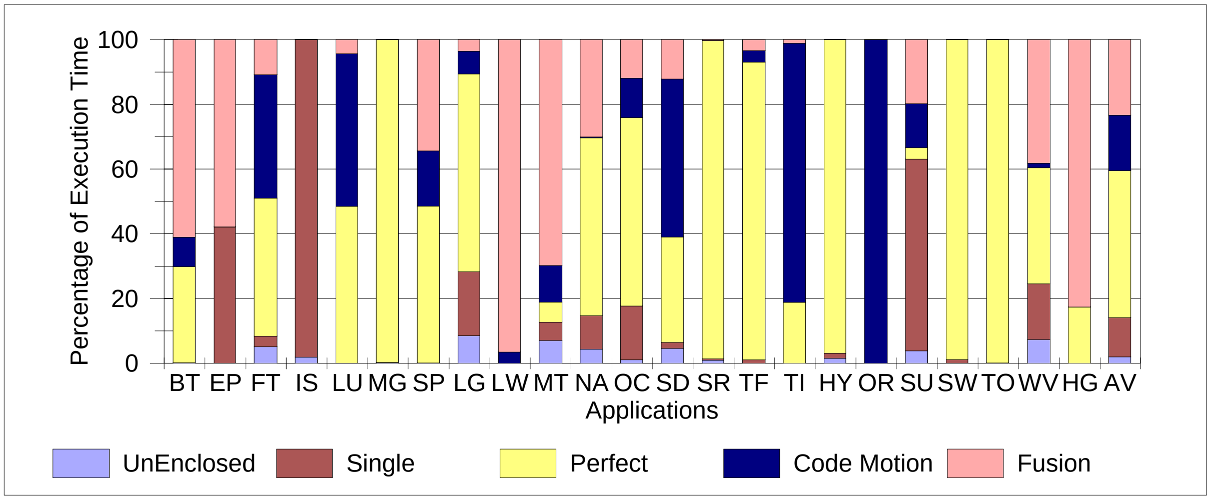
Figure 4.11(a) and Figure 4.11(b) show the characterization of execution time before and after loop distribution, respectively. These two figures indicate that in 9 of 12 applications, loop distribution only targets the execution time labelled “Code Motion”. In these applications, execution time labelled “Fusion” is not affected due to dependences. Only in MG, loop distribution succeeds in transforming all the imperfect nests, whose execution time is characterized as “Fusion”, to perfect nests. In SD and TF, loop distribution targets loop nests whose times are characterized as either “Code Motion” or “Fusion”. For most applications, most of the loops nests that can be made perfect by loop distribution have their execution time labelled as “Code Motion”, hence they can also be made perfect by code sinking. Loop nests that require loop fusion to become perfect, can rarely be made perfect by loop distribution.

Loop distribution increased the relative contribution to execution time of perfect nests in 12 benchmark applications. Figure 4.12 shows the increase in the relative contribution to execution time of pseudo and real perfect perfect nests, for the applications that are targeted by loop distribution.

As mentioned before, loop distribution only targets nests that can be made perfect by code sinking or loop fusion. Therefore, loop distribution has a potential of increasing

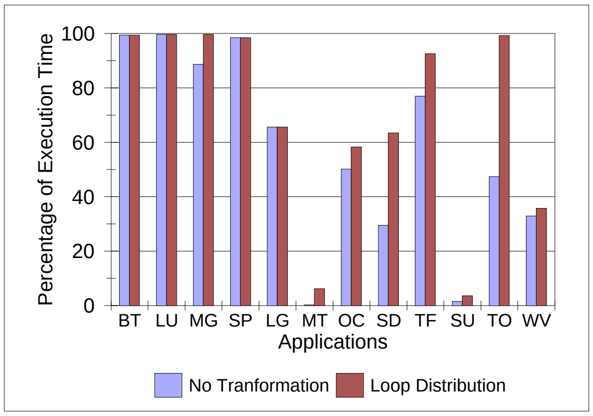


(a) Original

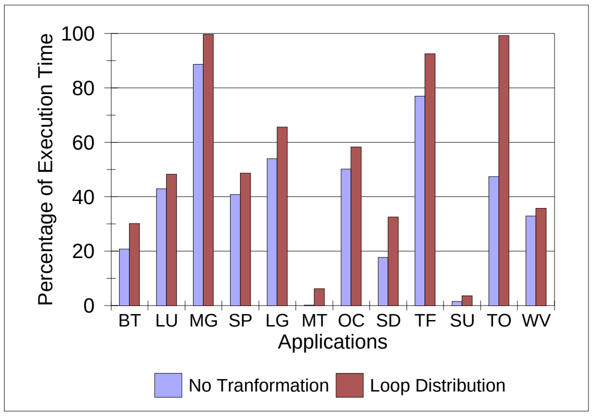


(b) After Loop Distribution

Figure 4.11: Characterization of execution time before and after loop distribution.



(a) Pseudo Perfect Nests



(b) Real Perfect Nests

Figure 4.12: Contribution to execution time of pseudo and real perfect nests before and after loop distribution.

the relative contribution to execution time of perfect nests in applications where the time classified under "Code Motion" and "Fusion" is significant (see Figure 4.11). Loop distribution succeeds in increasing the relative contribution to execution time of perfect nests to more than 95%, for 3 applications, namely MG, TF and TO. For 9 applications (other than the 3 above), loop distribution increases the relative contribution to execution time of perfect nests. However, in these 9 applications, there are loop nests which loop distribution has the potential to make perfect but fails to do so, as indicated below. For 7 applications, loop distribution is not effective in transforming imperfect nests into perfect ones.

Table 4.2 presents the reasons why loop distribution is ineffective in transforming more imperfect nests into perfect ones. The table shows that array dependences are often the main reason for disallowing distribution. However, it also shows that for FT, LU, OC, SD and TI, scalar dependences prevent loop distribution from obtaining perfect nests. Thus, scalar expansion has the potential of making loop distribution more effective for these applications. Unsafe loops disallow distribution in only few applications, namely, EP, MT and OR.

The applications in which the relative contribution to execution time of perfect nests increased due to loop distribution, are executed after applying loop distribution to assess the overhead introduced. It is found that loop distribution added negligible overhead to the execution time in all of the applications.

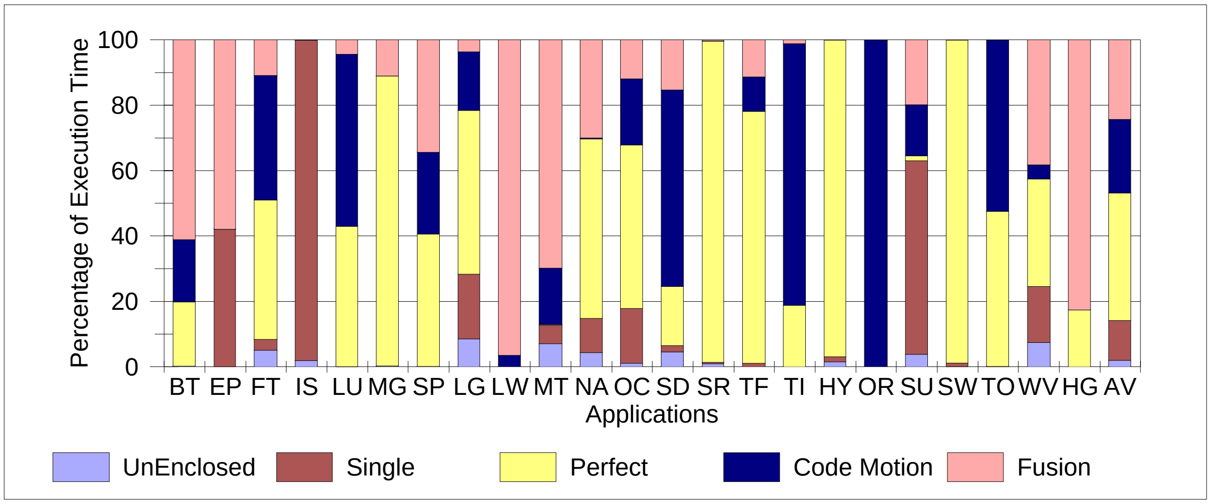
App	Total Loops	#Unsafe	%Time Unsafe	#Scalar Dep	%Time Scalar Dep	#Array Dep	%Time Array Dep
BT	193	7	6	2	0	25	63
EP	13	1	58	0	0	0	0
FT	63	4	8	6	31	3	9
LU	169	7	2	16	11	17	32
SP	247	9	3	6	0	27	46
LG	193	12	2	0	0	18	4
LW	106	0	0	4	0	17	96
MT	102	6	23	0	0	17	59
NA	246	13	8	5	0	18	21
OC	272	10	1	5	2	33	20
SD	246	26	15	18	39	7	8
TI	133	4	0	21	53	25	29
OR	7	2	100	0	0	0	0
SU	173	9	6	0	0	19	27
WV	566	2	0	14	1	43	39
HG	395	4	1	0	0	22	81

Table 4.2: The reasons for the ineffectiveness of loop distribution

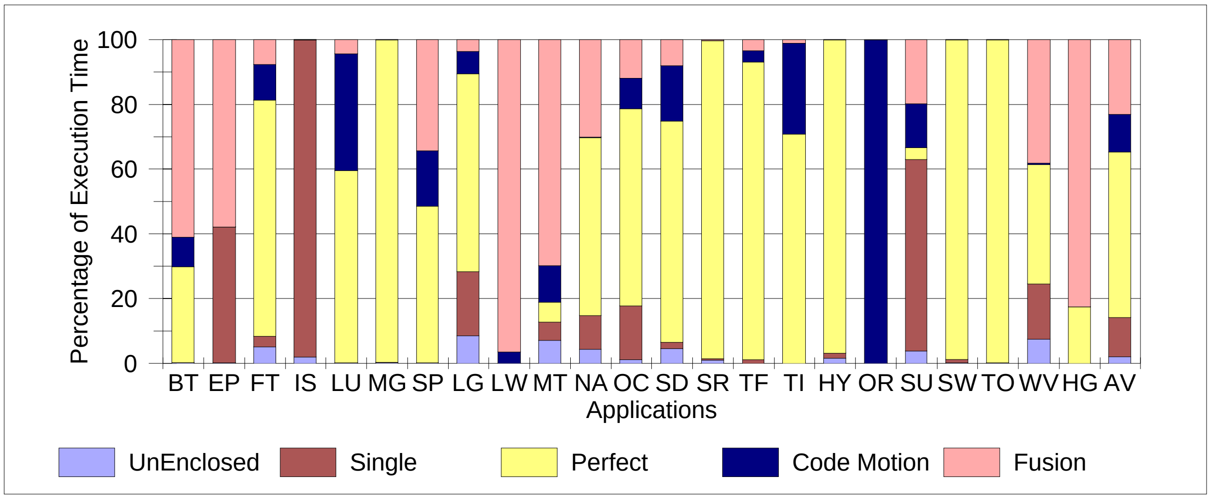
In summary, loop distribution succeeds in increasing the number and relative contribution to execution time of perfect nests in half of the applications. Array dependences are the main reason for the ineffectiveness of loop distribution in tranforming an imperfect nest into perfect one. However, in some applications, scalar dependences play a significant role in preventing distribution. In these cases, scalar expansion may be used to enable distribution.

4.4.2 Loop Distribution with Scalar Expansion

Loop distribution with scalar expansion targets the applications for which scalar dependences play a significant role in disallowing loop distribution from obtaining perfect nests. Figure 4.13(a) and Figure 4.13(b) show the characterization of execution time before and after loop distribution with scalar expansion, respectively. Loop distribution with scalar expansion mainly targets the execution time labelled as "Code Motion". Only FT and

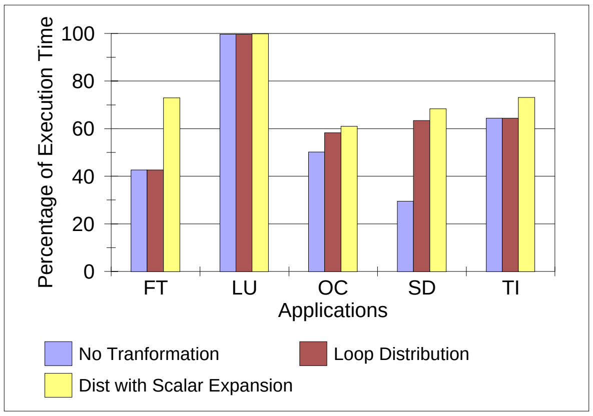


(a) Original

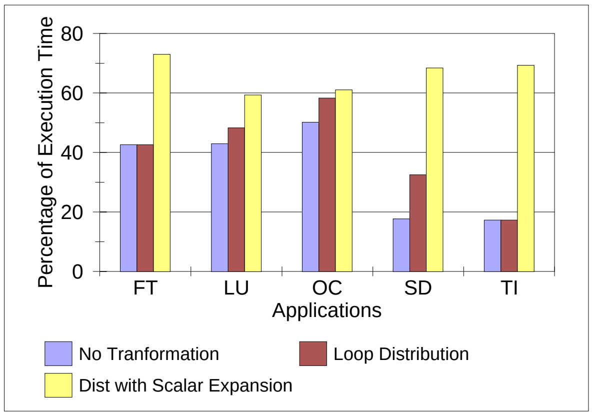


(b) After Loop Distribution with Scalar Expansion

Figure 4.13: Characterization of execution time before and after loop distribution with scalar expansion.



(a) Pseudo Perfect Nests



(b) Real Perfect Nests

Figure 4.14: Contribution to execution time of pseudo and real perfect nests before and after loop distribution with scalar expansion.

SD have their execution time labelled “Fusion” reduced by loop distribution with scalar expansion. Thus, loop nests, that require loop fusion to become perfect, can not be made perfect by loop distribution with or without scalar expansion.

Figure 4.14 shows the increase in the relative contribution to execution time of pseudo and real perfect perfect nests, for the applications targeted by loop distribution with scalar expansion, namely FT, LU, OC, SD and TI. For all five applications, scalar expansion makes loop distribution more applicable. Scalar expansion enables loop distribution for most of the nests in which scalar dependences disallow loop distribution. Thus, scalar expansion is legal in most of the cases where it is desirable.

4.4.2.1 Array Sizes of Expanded Scalars

To handle loops with variable trip counts, scalars are expanded into dynamically allocated arrays. Static arrays can only be used to expand scalars in loops with trip counts known at compile time[4]. It is found that most arrays must be allocated dynamically in four (FT, LU, SD, TI) of the five applications. Only in OC, the sizes of the arrays to be allocated are known at compile time.

[4]Theoretically, static arrays may be used for trip counts that are constant but unknown at compile time. However, the compiler used does not provide a mechanism for specifying static array sizes at run time.

Figure 4.15, shows the distribution of the sizes of the arrays allocated to expand the scalars in FT, LU, OC, SD and TI. For SD, some array sizes are omitted because they are used a very small number of times compared to the other sizes. For TI, array sizes are grouped in ranges; otherwise there would be too many array sizes to be shown on the horizontal axis of the figure. For instance, the figure indicates that 10 arrays of sizes ranging between 100kbytes and 1Mbytes are allocated. From the figure, two extreme cases can be seen. First, there is a large number (in the 100000's range) of small arrays (in the few hundred bytes range) to be allocated, as for FT and SD. In this case, the overhead is expected to be due to the large number of calls to the memory management routines. Second, there may be a small number (in the 10's range) of large arrays (more than hundred kilobytes) to be allocated, as for LU and TI. In this case, the overhead is expected to be mainly due to the large arrays allocated and their effects on the cache. These large arrays would occupy most of the cache and thus increase the number of cache misses to other data. The large size of some of the arrays (e.g. in TI) are due to applying scalar expansion in loop nests, where the array size must be equal to the multiplication of the trip counts of the loops in the nests (Section 3.5).

4.4.2.2 Overhead of Scalar Expansion

The overhead of scalar expansion has multiple sources. First, references to arrays that replace scalars may cause cache misses to original data. Second, references to arrays must go to the cache while references to scalars can be satisfied by a register. Third, there is a computational overhead in evaluating the subscript expressions of the array references. Fourth, when using dynamically allocated arrays, memory management routines such as *malloc* and *free* increase execution time.

Figure 4.16 shows the overhead of applying loop distribution with scalar expansion to the five applications. The numbers in Figure 4.16 are normalized to the execution time of the applications without any transformations:

- For TI, scalar expansion caused an overhead of 250%. This huge overhead is due to cache misses caused by the large sized arrays allocated. The allocation of very large chunks of memory (in the order of 1 Mbytes to 20 Mbytes, see Figure 4.15)

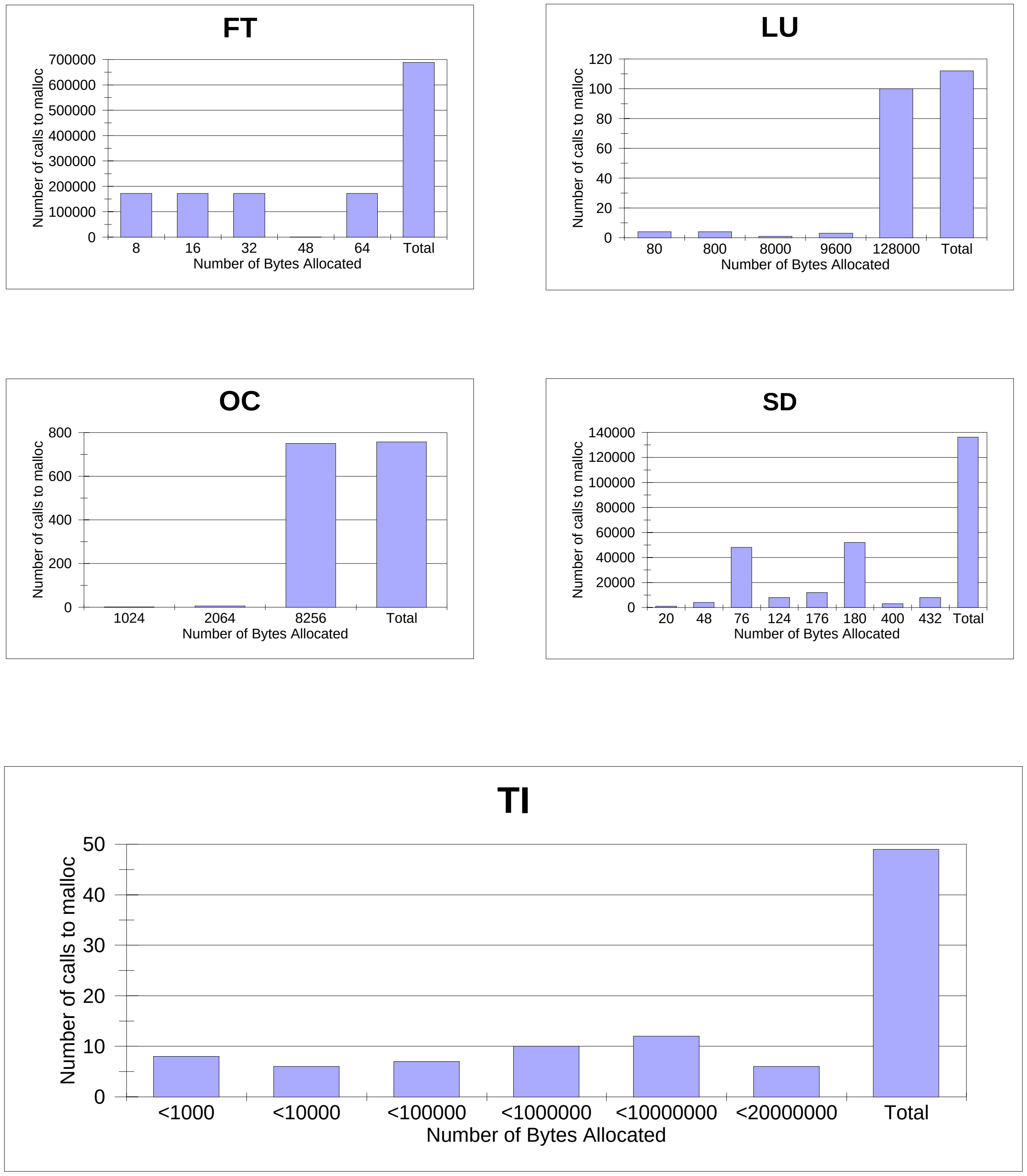


Figure 4.15: Size of Arrays Allocated in Scalar Expansion

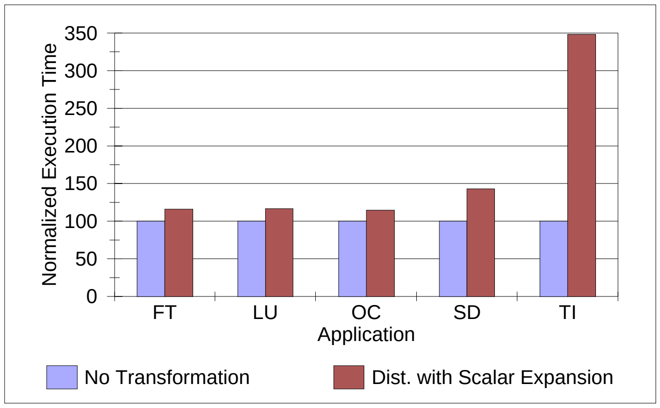


Figure 4.16: Overhead of Loop Distribution with Scalar Expansion

also contribute to the overhead.

- For FT and SD, scalar expansion caused overheads of 16% and 43%, respectively. These overheads are caused by a large number (more than 100000) of calls to *malloc* even though the size of the allocated arrays are relatively small (see Figure 4.15).

- For LU and OC, the overheads of scalar expansion are 17% and 15%. The overheads are due cache misses introduced by accessing the arrays introduced by scalar expansion. Also, the overhead is due to several hundreds of calls to *malloc* to allocate arrays in the kilobytes size range.

The overhead due to scalar expansion is prohibitive, and thus scalar expansion should be applied with special care. It is likely to be useful when the sizes of the arrays to be allocated and the number of calls to malloc are small.

4.5 The Effect of Loop Fusion on Nest Structures

This section reports on how loop fusion affects the contribution of the perfect nests in the benchmark applications. It also discusses the overhead introduced by loop fusion.

Loop fusion increases the relative contribution to execution time of perfect nests in only 2 applications. The relative contribution to execution time of real and pseudo perfect nests increases from 88% to 99% in MG. In SD, the relative contribution to execution time of pseudo perfect nests increased from 29% to 34%, and the relative contribution to execution time of real perfect nests increased from 18% to 23%. Loop fusion caused no overhead in MG, and a 3% overhead in SD.

Loop fusion has the potential to increase perfect nests contribution in the applications that have a significant portion of its execution time labelled as "Fusion" in Figure 4.7. For these applications, the reasons why loop fusion failed to increase perfect nests are presented in Table 4.3. MG is omitted because loop fusion brought the relative contribution to execution time of perfect nests to 99%. The main reason for the ineffectiveness of loop fusion is the presence of backward dependences in the fused loop which makes fusion illegal. The presence of INTERCODE (i.e. code between loop nests) that cannot be moved is a factor in OC only.

App	**Total Loops**	**#Unsafe**	**%Time Unsafe**	**#Intercode**	**%Time Inter-code**	**#Back Dep**	**%Time Back Dep**
BT	205	2	3	0	0	12	59
EP	13	1	58	0	0	0	0
FT	63	2	3	0	0	3	7
SP	247	6	2	0	0	15	33
LW	106	0	0	2	1	5	96
MT	102	4	17	1	1	13	49
NA	246	3	2	0	0	4	26
OC	272	6	1	4	4	9	7
SD	246	12	5	1	0	7	6
SU	173	9	6	0	0	12	13
WV	566	2	0	0	0	23	38
HG	395	2	1	0	0	22	81

Table 4.3: The reasons for the ineffectiveness of loop fusion

4.6 Summary

Code sinking, loop distribution and loop fusion are three transformations that can transform an imperfect nest into a perfect one. These three transformations, along with loop distribution with scalar expansion are implemented and applied to a suite of 23 applications. Loop fusion is found to be effective at transforming imperfect nests into perfect ones in only two applications (see Section 4.5). The effects of the three other transformations are summarized below, using three criteria: the number of perfect nests, the relative contribution to execution time of perfect nests and the overhead introduced.

The effects of the transformations on the numbers and dimensions of perfect nests are shown in Figure 4.17. The figure shows that transformations— that require perfect nests— have to deal mostly with perfect nests with small dimensions (typically 2 or 3) even after increasing the number of perfect nests. Therefore, contrary to previous assumptions [KM92], algorithms that exhaustively try different loop orderings of a nest, would not be impractical. The maximum dimension of a perfect nest is 2 or 3, in 9 of the 14 applications. For 4 of 14 applications, BT, LU, SP and SU, the maximum dimension of a perfect nest is 4. Only in LG, there is one perfect nest with a dimension of 5. Also in all of these applications except MG, perfect nests with dimension equal to 2, constitute the majority (more than 60%) of perfect nests. Moreover, the number of perfect nests decreases with the increase of the dimension.

In 7 of 14 applications (BT, LU, MG, SP, SD, TF and TI), loop distribution with scalar expansion results in more perfect nests than code sinking. The reason is that the latter transformation transforms one imperfect nest into one perfect nest, while the former transformation splits loop nests into multiple nests that may or may not be perfect.

Altogether, the three transformations increase the relative contribution to execution time of perfect nests in 14 benchmarks. The effects of the three transformations on the relative contribution to execution time of pseudo and real perfect nests are shown in Figures 4.18 and Figure 4.19, respectively. The average relative contribution to execution time of real perfect nests for the 14 applications can be increased from 38% to 66% by code sinking, to 58% by loop distribution with scalar expansion, and to 50% by loop distribution without scalar expansion. It should be noted that scalar expansion made loop distribution more effective in 5 applications.

The transformations used to increase the perfect nests introduce some overhead to the execution time of the applications. Loop distribution is found to have no overhead. Code sinking caused no overhead in more than half of the applications. It caused low overhead (less than 5%) in some applications. For few applications, the overhead of code sinking is found prohibitive (more than 10%). Scalar expansion caused the highest overhead (more than 15%) in the applications it targeted.

To conclude, given an imperfect nest that needs to be made perfect, loop distribution should be tried first because it causes the lowest overhead. If loop distribution fails to make the nest perfect, then code sinking should be tried next because it introduces a smaller overhead than loop distribution with scalar expansion. If code sinking failed to make the nest perfect then loop distribution with scalar expansion could be tried. However, special attention should be paid, when applying scalar expansion, especially to the size of the arrays to be allocated.

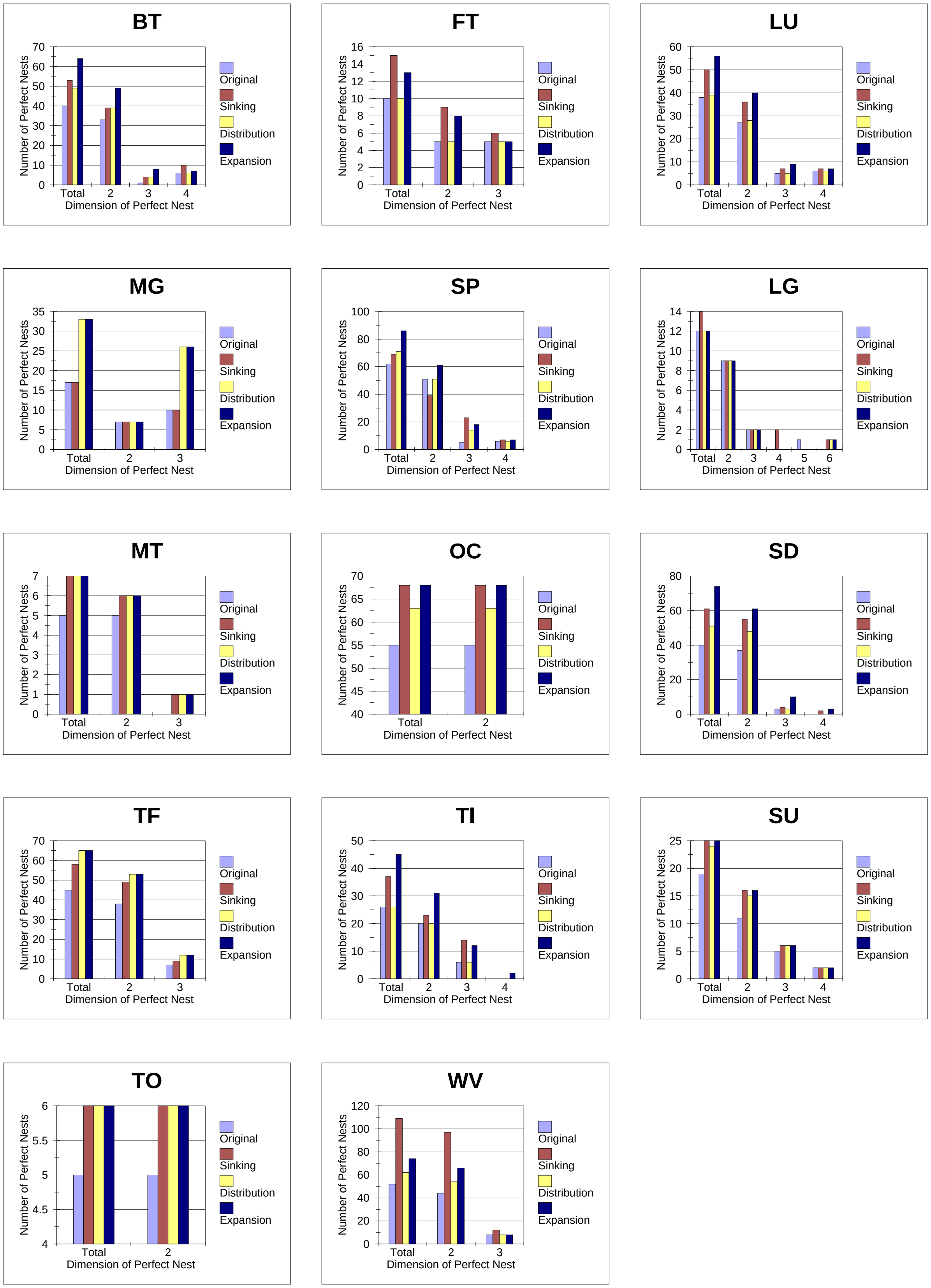


Figure 4.17: Variation of the numbers and dimensions of perfect nests before and after the application of code sinking, loop distribution, and loop distribution with scalar expansion.

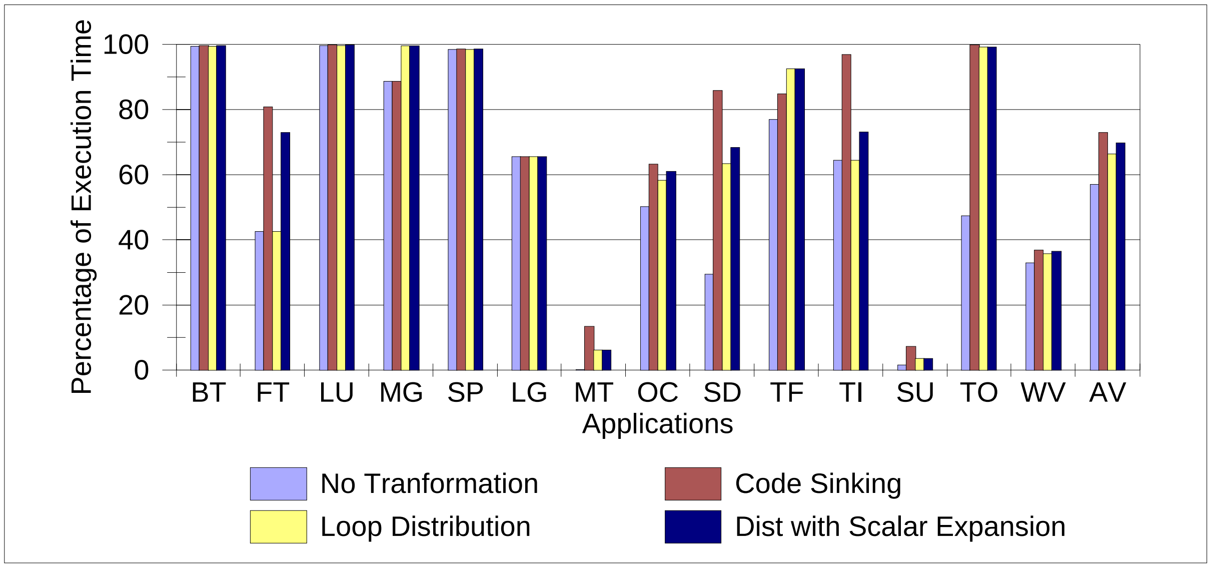


Figure 4.18: Contribution to execution time of pseudo perfect nests before and after code sinking, loop distribution and loop distribution with scalar expansion.

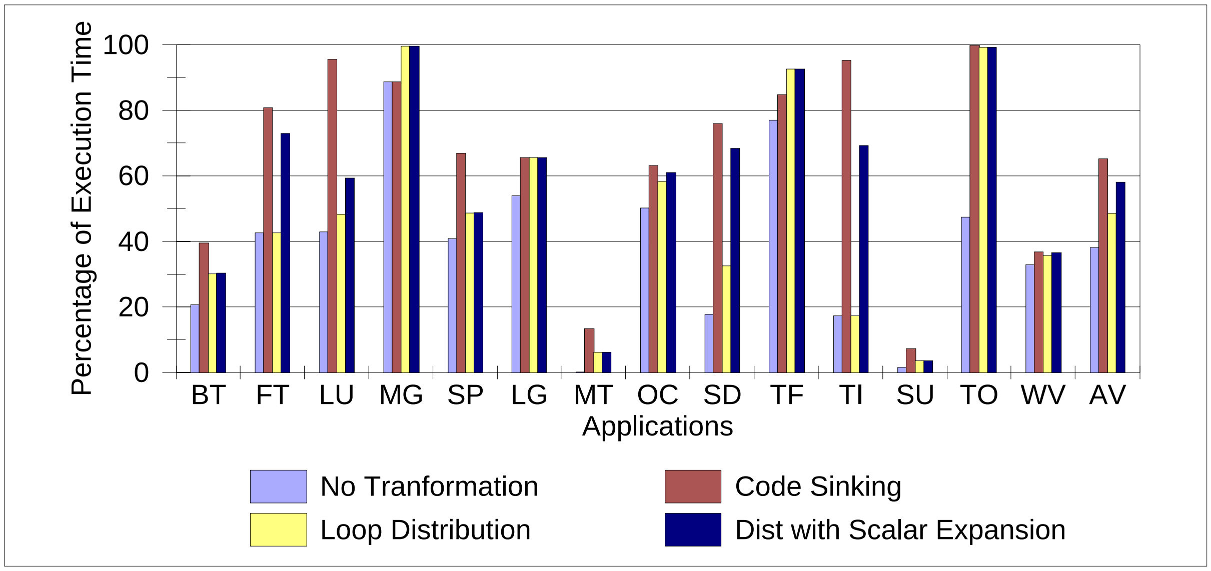


Figure 4.19: Contribution to execution time of real perfect nests before and after code sinking, loop distributionand loop distribution with scalar expansion.

Chapter 5

Improvements in Locality

This chapter presents experimental results concerning cache locality. Section 5.1 considers the applicability of loop permutation and how it is affected after applying the techniques that increase the relative contribution to execution time of perfect nests. Similarly, Section 5.2 discusses the potential benefit of tiling and how it is affected after applying the techniques that increase the relative contribution to execution time of perfect nests. Section 5.3 presents the results on the need to enhance spatial locality.

5.1 Applicability of Loop Permutation

Loop permutation is the most commonly used techniques to enhance cache locality [CMT94]. However, loop permutation targets only perfect nests. This section presents results on how applicable is loop permutation, and on how its applicability is affected by the transformations discussed in the previous chapter.

A nest N is classified as *permutable* if it is perfect and there exists at least a pair of loops belonging to N that can be permuted[1]. The relative contribution to execution time of permutable nests is measured before and after code sinking, loop distribution, and loop distribution with scalar expansion. Loop fusion is not considered because it is ineffective in increasing perfect nests. Figure 5.1 shows the relative contribution to execution time of permutable perfect nests for the 23 applications considered. The relative contribution to execution time of permutable nests is negligible in quarter of the applications, EP, IS, LW, MT, OR and SU. This is expected because Figure 4.7, in the previous chapter,

[1]The dependences are examined to determine if permutation is legal.

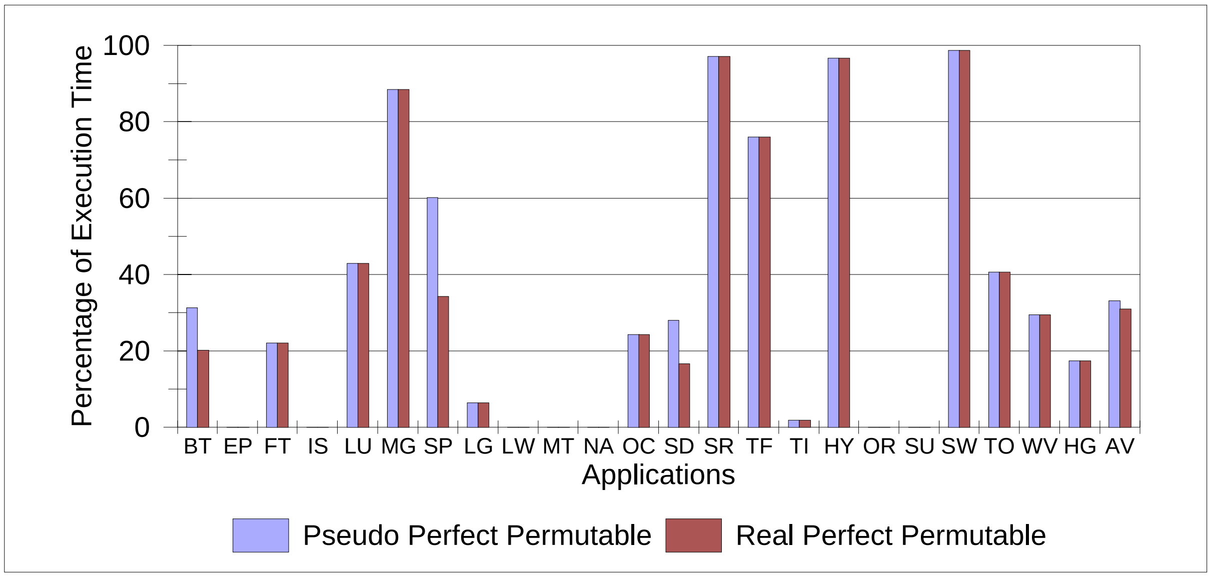


Figure 5.1: Contribution to execution time of permutable pseudo and real perfect nests.

shows that for these applications, perfect nests account for less than 1% of execution time. The relative contribution to execution time of permutable perfect nests in half of the applications (BT, LU, MG, SD, SR, TF, HY, SW, TO, WV and HG) is equal to the contribution to execution time of perfect nests. This implies that in these applications, all the perfect nests with significant contribution to execution time are permutable. For the remaining quarter of the applications, FT, SP, LG, NA, OC and TI, permutable perfect nests contribute less than perfect nests to execution time. Moreover, in most (20) applications, pseudo and real permutable perfect nests have identical relative contribution to execution time.

Figure 5.2 and Figure 5.3 show the effects of the three transformations on permutable pseudo perfect nests and permutable real perfect nests[2], respectively. All the applications considered in the last two figures have shown an increase in the contribution to execution time of perfect nests. In most applications (11 of 14) the increase in the relative contribution to execution time of perfect nests causes an increase in the relative contribution to execution time of permutable nests. In less than a quarter of the applications (3 of 14, FT, LG and SU), the increase in the relative contribution to execution time of perfect nests does not cause an increase in the relative contribution to execution time of permutable

[2]Dependence testing is rerun on the transformed applications to determine which nests are permutable.

nests. The previous chapter shows that code sinking is the most effective in increasing the contribution to execution time of real perfect nests. In fact, code sinking achieved the highest contribution to execution time of real perfect nests for 12 of 14 applications (except for MG and TF). For permutable perfect nests, however, Figure 5.3 shows that code sinking is more effective than loop distribution for only 5 applications (BT, LU, SP, MT and OC). Loop distribution is more effective than code sinking for 3 applications (MG, SD and TF). For the other six applications, the two transformations achieve similar results. When applying scalar expansion, loop distribution achieves better results than code sinking for LU, MG, SD, TF and TI. Considering the averages, it can be seen that code sinking achieves similar results as loop distribution. However, when applying scalar expansion to enable loop distribution, the average contribution to execution time[3] of permutable nests is higher than the one obtained by code sinking. The reason for this is that code sinking creates perfect nests with more complex dependences than does loop distribution. This should be expected since code sinking pushes additional statements into the body of a loop, while loop distribution moves these statements to a separate loop. Moreover, scalar expansion eliminates scalar anti and output dependences, and replaces scalar flow dependences with loop-independent dependences, hence simplifying the dependence relations in the nest.

5.2 Potential Benefit of Tiling

This section reports on the potential benefit of tiling which is one of the most commonly used techniques for enhancing temporal locality. A loop nest is considered *tilable* if it is real perfect and encloses at least an array reference with a number of variable subscripts[4] (referred to later as variable dimension) smaller than the dimension of the nest. The presence of this property (tilable) indicates that part of the array is being iterated over, hence there will be temporal locality. Tiling may also be beneficial in exploiting spatial locality [BGS94]. However, we believe that tiling is mostly effective for temporal locality,

[3]The overhead of scalar expansion is not included.

[4]The number of variable subscripts in an array reference, or the variable dimension of the array reference, is equal to the total number of subscripts minus the number of subscripts with constant expressions.

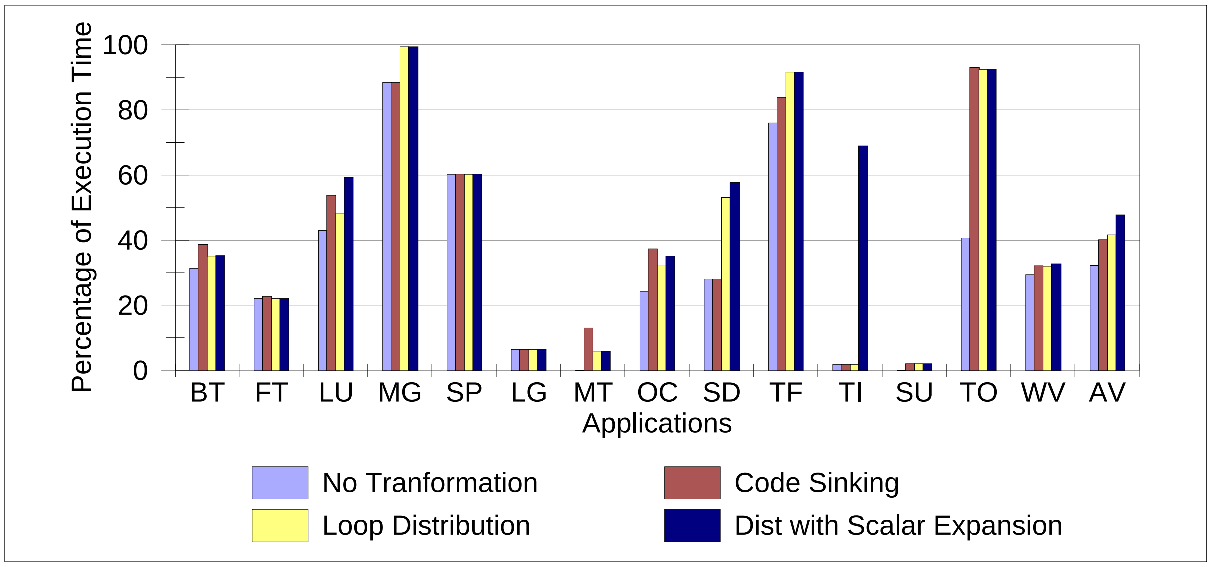


Figure 5.2: Contribution to execution time of permutable pseudo perfect nests before and after code sinking, loop distribution, loop distribution with scalar expansion.

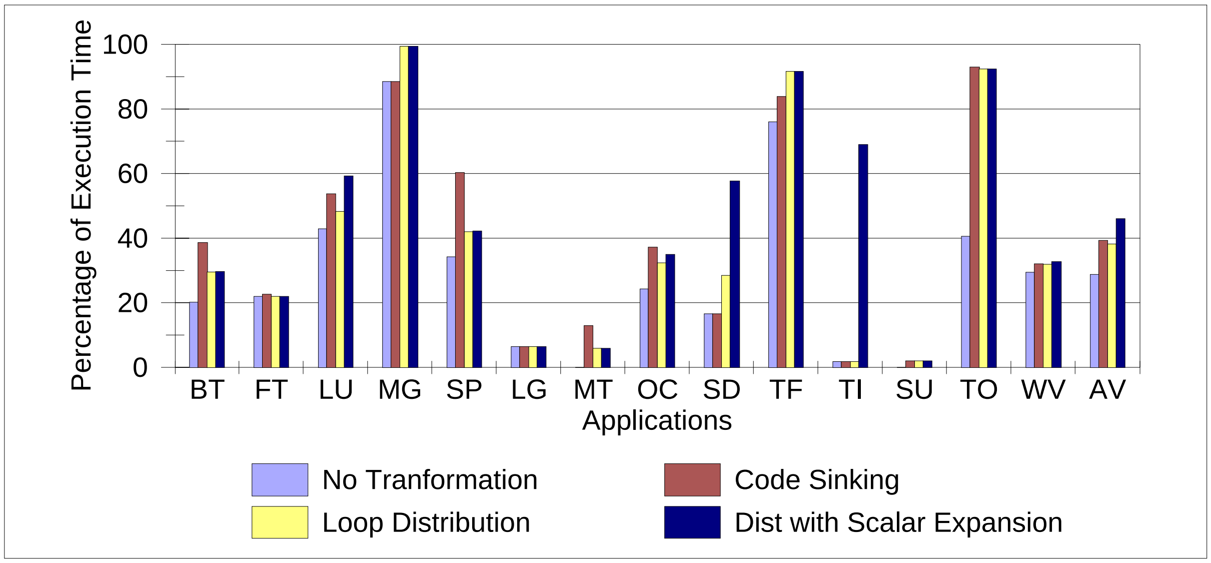


Figure 5.3: Contribution to execution time of permutable real perfect nests before and after code sinking, loop distribution, loop distribution with scalar expansion.

which is reflected by our definition of tilable nests.

When approximating the potential benefit of tiling , two options are considered.

- **Binary:** if the nest is tilable then the time for the whole nest is considered as part of the potential benefit of tiling.

- **Linear:** given a tilable nest where there are `n` array references in total, and `m` array references with variable dimensions smaller than the dimension of the nest, then the time considered as part of the potential benefit of tiling is the whole time of the nest, scaled by $\frac{\mathtt{m}}{\mathtt{n}}$.

Figure 5.4 shows the relative contribution to execution time of tilable nests using the binary and linear approximations, for all the applications considered. Both approximations give the same results for 20 applications. Thus, in most applications, when one of the array references has a variable dimension that is smaller than the dimension of the enclosing nest, then most array references in the same nest also have a variable dimension that is smaller than the dimension of the nest. For 17 applications, tilable nests contribute to less than 3% of execution time. This is expected for 6 applications (EP, IS, LW, MT, OR snd SU) because in these applications perfect nests have relative contribution to execution time of less than 2%. For 11 applications (BT, LU, MG, SP, LG, SD, TF, TI, TO, SW and SR), tilable nests have relative contribution to execution time to execution time less than 3%, even though perfect nests in these applications contribute significantly (more than 20%) to execution time. For the remaining 6 applications, namely FT, NA, OC, HY, WV and HG, tilable nests contribute significantly (between 10% and 50%) to execution time.

Thus, tiling may be beneficial to a quarter (6 of 23) of the applications. Tiling is not beneficial for another quarter (6 of 23) of the applications because perfect nests in these applications have negligible contribution to execution time. For the remaining half (11 of 23) of the applications, tiling is not beneficial because the perfect nests in these applications do not have temporal locality that is targetable by tiling as defined in this research.

Figure 5.5 and Figure 5.6, show the effects of code sinking, loop distribution, and loop distribution with scalar expansion on the potential benefits of tiling using the binary and linear approximations, respectively. Only the applications whose relative contribution to execution time of perfect nests increased due to any of the transformations are considered. When deriving the results the following two points are considered. First, the array references, whose variable dimensions become smaller than the dimension of their enclosing nest because they are sunk inside a loop, are not considered when determining whether a nest is tilable. Second, the array references introduced by scalar expansion are not considered when determining whether a nest is tilable. Increasing the relative contribution to execution time of perfect nests does not increase the potential benefit of tiling in a third (5 of 14) of the applications, BT, MG, SP, TF and TO. The contribution to execution time of tilable nests in these applications remains negligible. This indicates that in these applications, the original perfect nests and those obtained by the three transformations do not have temporal locality that is targetable by tiling.

Increasing the relative contribution to execution time of perfect nests increases the potential benefit of tiling in two thirds (9 of 14) of the applications. This indicates that in these applications, there are imperfect nests for which tiling is desirable. Moreover, these imperfet nests can be made perfect by one of the three transformations.

Figures 5.5 and 5.6 show that loop distribution with scalar expansion achieves higher potential benefit for tiling than code sinking, for SD and TI. However, Figures 4.9 and 4.14, in the previous chapter, show that code sinking achieves higher relative contribution to execution time of perfect nests than loop distribution with scalar expansion for the same two applications. Figure 4.17 shows that loop distribution with scalar expansion results in more perfect nests than code sinking for SD and TI. The additional perfect nests are tilable nests and account for the difference between the potential benefits of tiling after code sinking and after loop distribution with scalar expansion.

Given that tiling mainly targets tilable nests as defined in this research, 6 of 23 applications would have a potential benefit from tiling. Applying code sinking or loop distribution with scalar expansion, makes 5 additional applications targetable by tiling. The transformations also increase the potential benefit of tiling for 3 of the 6 applications that may benefit from tiling originally.

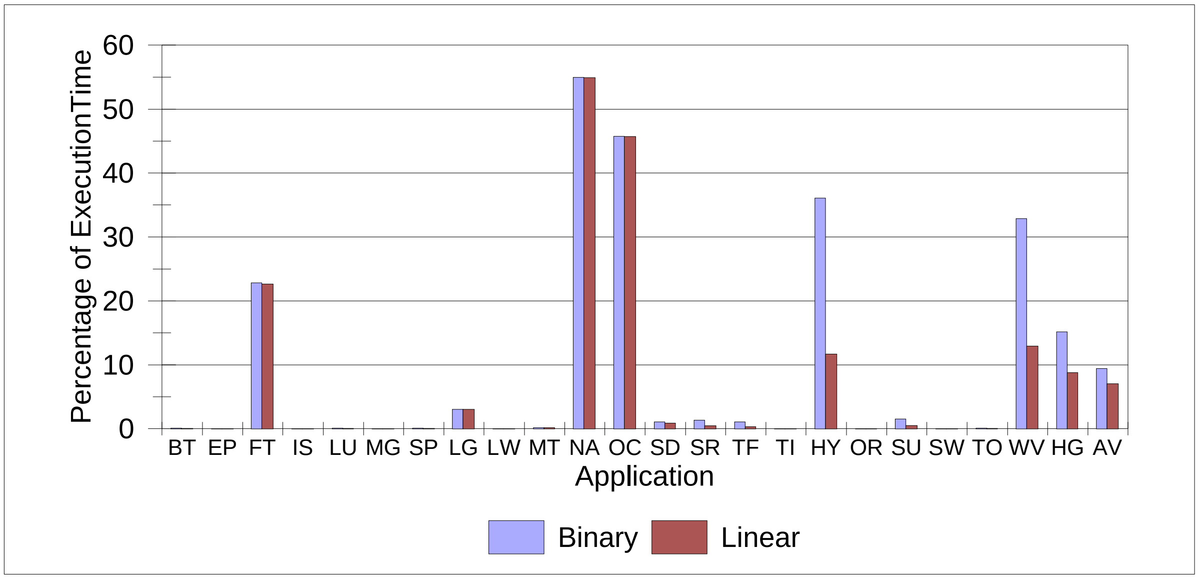


Figure 5.4: Contribution to execution time of tilable nests for both binary and linear approximations.

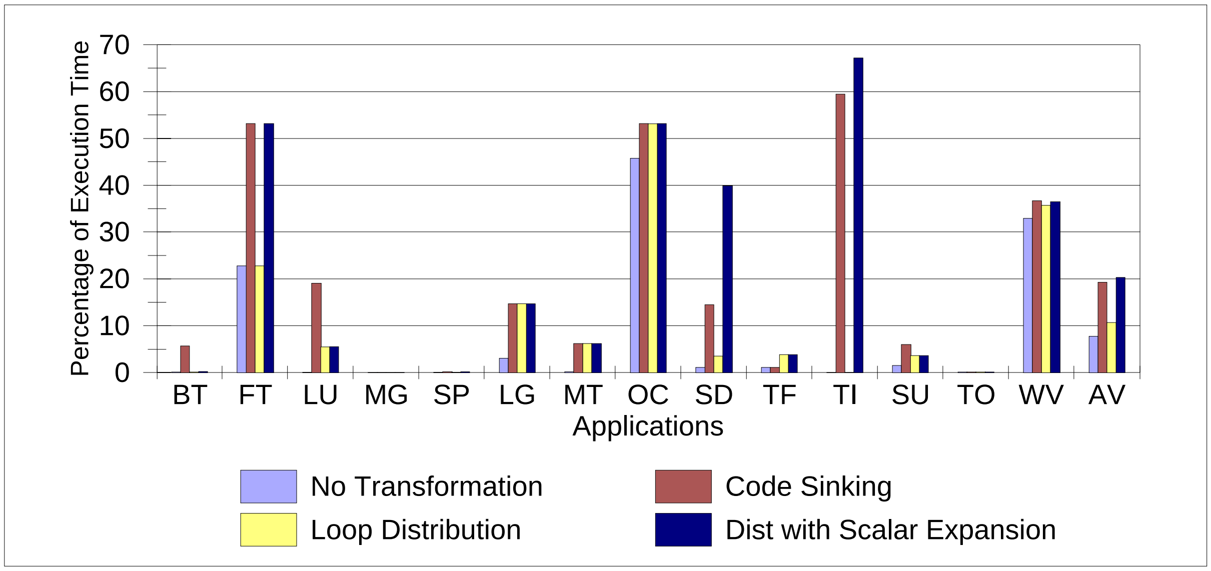


Figure 5.5: Contribution to execution time of tilable (binary) perfect nests before and after code sinking, loop distributionand loop distribution with scalar expansion.

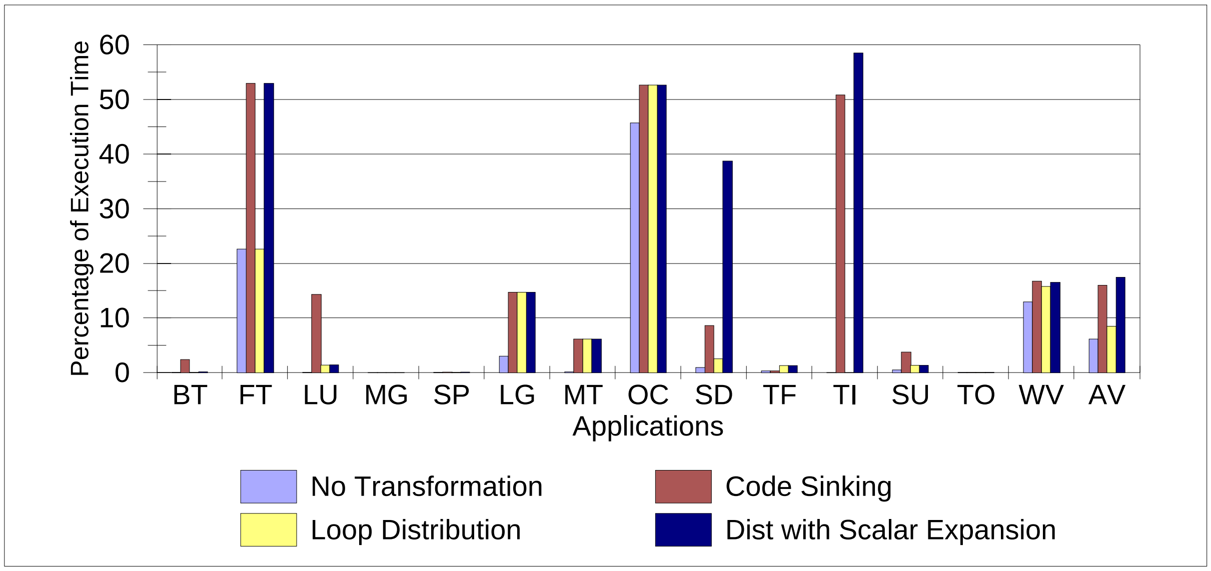


Figure 5.6: Contribution to execution time of tilable (linear) perfect nests before and after code sinking, loop distributionand loop distribution with scalar expansion.

5.3 Improvements in Spatial Locality

This section reports on the need for loop permutation and flexible array layout to enhance cache spatial locality. First, the section describes the characteristics of individual array references with respect to self-spatial locality and group-spatial locality. Then, it evaluates the need for flexible array layout to enhance spatial locality. Finally, it examines the need for loop permutation to improve spatial locality.

5.3.1 Characteristics of Array References

This section reports on the characteristics of array references regarding self-spatial locality and group-spatial locality. Conflicts among array references enclosed by the same nest regarding the best innermost loop, and conflicts among array references to the same array regarding the best array layout, are not considered in this section; they will be considered later.

5.3.1.1 Array References and Self-Spatial Locality

All array references are examined and classified under the categories discussed in Section 3.7.1.1. Figure 5.7 shows a breakdown of the array references of different applications. Figure 5.8 is similar to Figure 5.7, however, it shows the relative contribution to

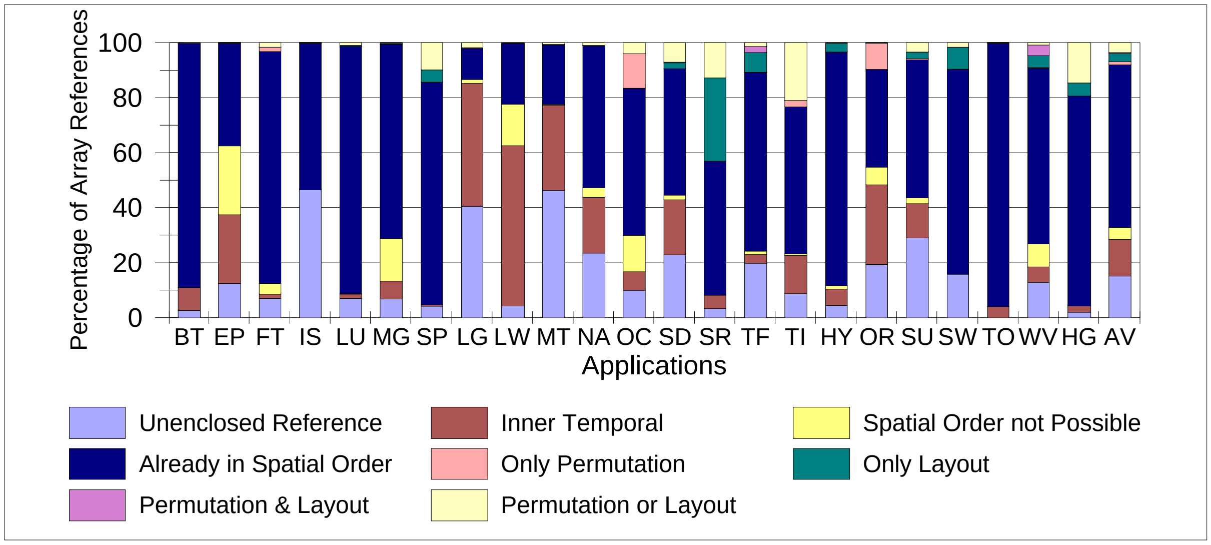


Figure 5.7: Classification of array references vis-a-vis what needs to be done to achieve self-spatial locality.

execution time of the different array reference classifications. The relative contribution to execution time of an array reference is *approximated* as follows.

- The relative contribution to execution time of an array reference is equal to the net execution time of the enclosing loop divided by the number of array references enclosed by this loop. The aim of this approximation is to assign different weights to array references in different loop nests, depending on the execution time of the of the nests involved. Therefore, we believe that this approximation is a reasonable one even though it ignores the effects of arithmetic operations.

- The relative contribution to execution time of an unenclosed array reference is equal to the net execution time (excluding loop times and unenclosed subroutine times) of the program unit divided by the number of unenclosed array references.

Averaged over all the benchmarks, unenclosed array references constitute 15% of all array references and contribute to only 1% of the total execution time. This is expected since loops contribute to more than 98% in most applications. Also, 60% of array references, contributing to 66% of the total execution time, already exhibit self-spatial locality. Moreover, array references, exhibiting self-temporal locality with respect to the loop di-

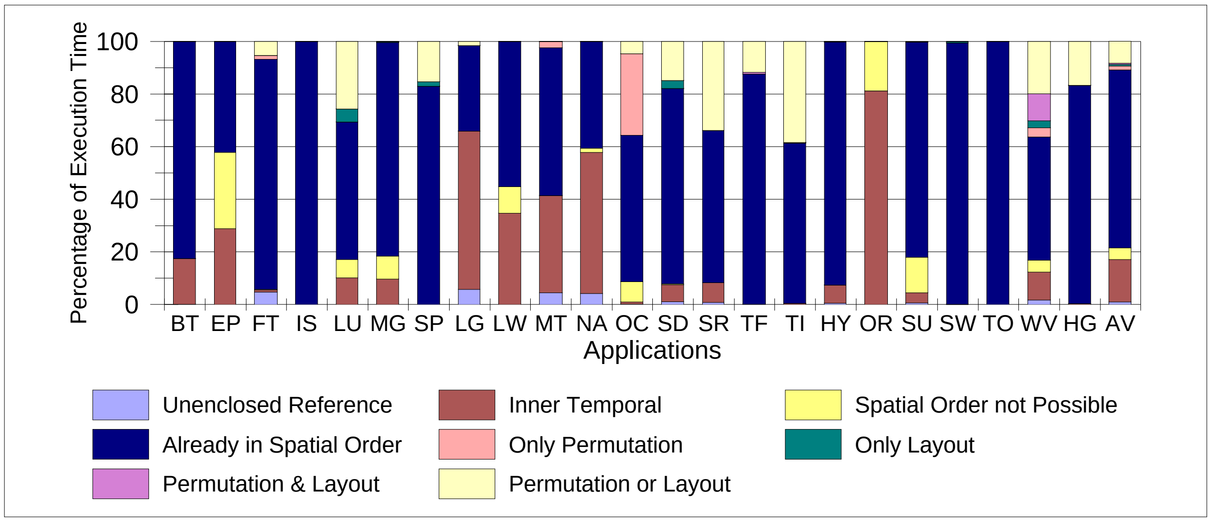


Figure 5.8: Classification of array references time vis-a-vis what needs to be done to achieve self-spatial locality.

rectly enclosing them, constitute 12% of all array references, and contribute to 16% of the total execution time. Hence, it can be concluded that program developers ensure that the majority of array references exhibit either self-temporal locality or especially self-spatial locality.

For more than half (13) of the applications, self-spatial locality cannot be improved by neither loop permutation nor by flexible data layout. In 11 of 13 of these applications, most of the array references already exhibit self-temporal locality or self-spatial locality. In the other two applications (EP and OR), the majority of array references have subscript expressions that prevent self-spatial locality. For 40% (9) of the applications, array references that can be made to exhibit self-spatial reuse contribute to a significant percentage of execution time. For these array references in these applications, either loop permutation or change in the array layout is applicable. Only in OC, there are array references that can be made to exhibit self-spatial locality using loop permutation only. Loop permutation and flexible data layout are not required together (except for WV) to make an array reference exhibit self-spatial locality.

Hence, in most applications, the majority of array references already exhibit self-spatial locality. When there is a potential to increase the number and the relative con-

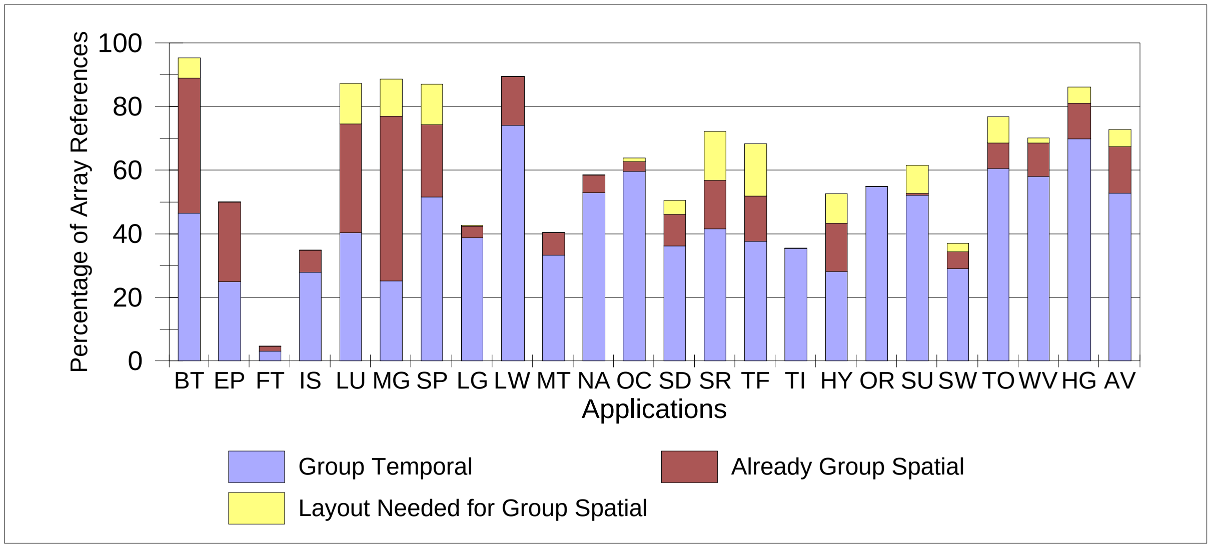


Figure 5.9: Classification of array references vis-avis group-spatial locality.

tribution to execution time of array references that exhibit self-spatial locality, either loop permutation or change in the array layout can be used. The case when only one of the two transformations can be used is rare. The case where both transformations are needed to benefit individual references is also rare.

5.3.1.2 Array References and Group-Spatial Locality

All array references are examined and classified under the categories discussed in Section 3.7.1.2. The categories reflect what needs to be done to achieve group-spatial locality. Figure 5.9 shows a breakdown of the array references of the 23 benchmark applications. Figure 5.10 is similar to Figure 5.9, however, it shows the relative contribution to execution time of the different array reference classifications.

Averaged over all the applications, most (52%) array references exhibit group-temporal reuse. Some (15%) of the array references already exhibit group-spatial reuse. Only (7%) of the array references can be made to exhibit group-spatial reuse by changing the array layout. The remaining array references cannot be made to exhibit group-spatial reuse. In half (12) of the applications, changing the array layout may increase the number of array references exhibiting group-spatial locality.

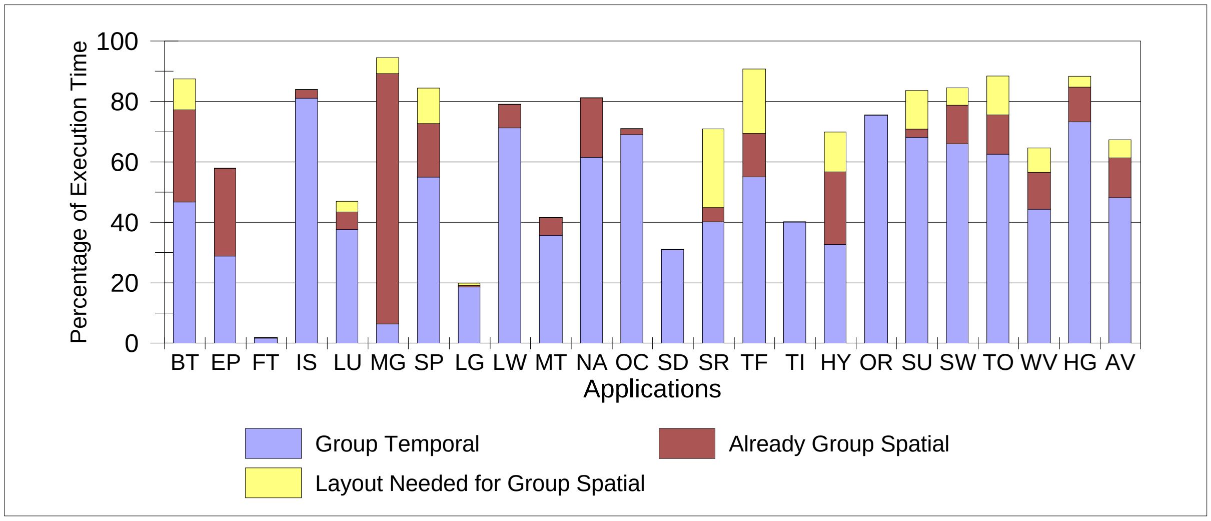


Figure 5.10: Classification of array references time vis-a-vis group-spatial locality.

5.3.2 Effect of Flexible Layout on Spatial Locality

This section presents results on whether having different layouts for different arrays in an application would be beneficial. The framework used to obtain the results is described in Section 3.7.2. The sections 5.3.1.1 and 5.3.1.2 above considered changing the layout to benefit an individual array reference, by making it exhibit self or group-spatial locality. However, the above sections do not consider conflicts in the array layouts among references to the same array. Figure 5.11 shows the percentage of array references that exhibit self or group-spatial locality when the array layout is fixed, and when it can be different for different arrays. Figure 5.12 is similar to Figures 5.11, however, it shows the contribution to execution time of references instead of the percentage of array references. The following remarks discuss the results.

- For 7 applications, EP, LG, LW, MT, NA, OC and OR, having different layouts for different arrays does not increase the references with spatial locality. This result can be explained by examining Figures 5.8 and 5.10. Since, Figures 5.8 and 5.10 show that for the 7 applications above, data layout cannot benefit individual array references, therefore when considering all references together, data layout would not be benefitial.

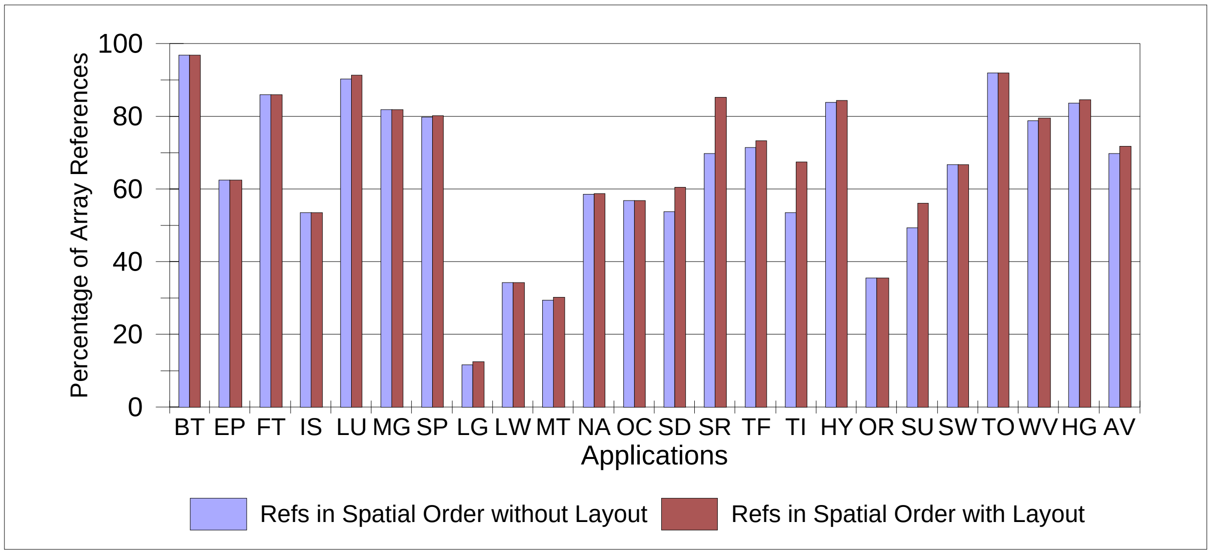


Figure 5.11: Effects of layout change on the spatial locality of array references

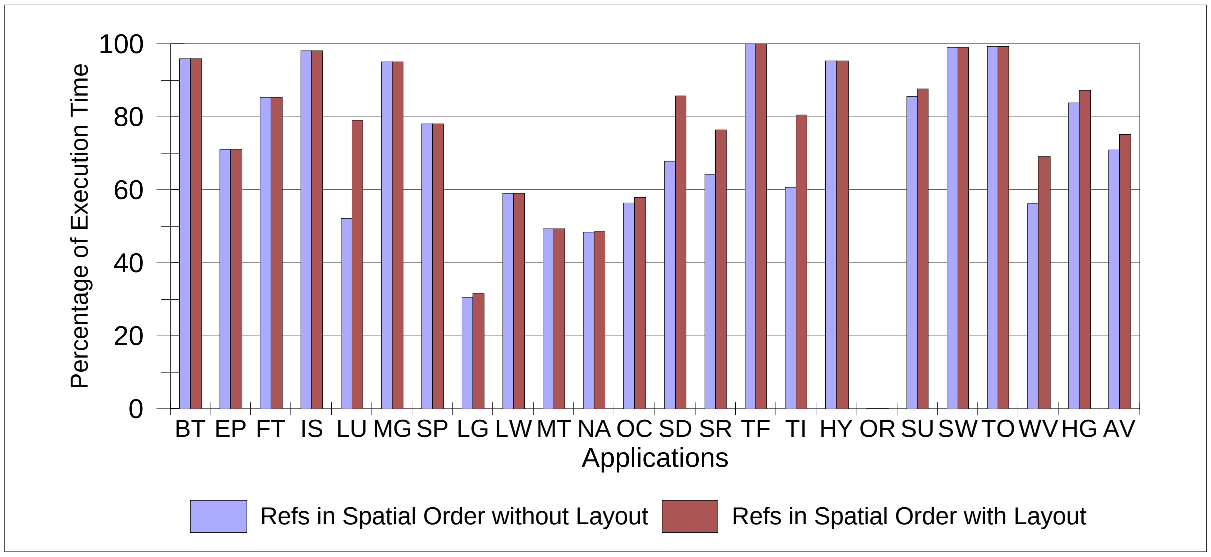


Figure 5.12: Effects of layout change on the spatial locality of array references

- For 6 applications, BT, FT, IS, HY, TO and SW, data layout does not increase the contribution to execution time of references with spatial locality. Figure 5.8 shows that the references with self-spatial locality contribute to more than 90% of execution time. Therefore, changing the layout to increase the references with group-spatial locality would decrease the references with self-spatial locality. Thus changing the data layout for these applications is not effective.

- For 4 applications, MG, SP, TF and SU, changing the data layout has the potential of making additional references exhibit spatial locality, as can be seen from Figures 5.8 and 5.10. However, due to conflicts, changing the data layout in these applications causes an overall decrease in array references exhibiting spatial locality.

- For only a quarter (6 of 23) of the applications (LU, SD, SR, TI, WV and HG), having different layouts for different arrays increases the overall number and contribution to execution time of the array references that exhibit either self or group-spatial locality. However, it should mentioned that once the legality of changing the array layout is taken into consideration then the benefits for these applications may be reduced.

In summary, it is found that changing the data layout of some of the arrays in 17 applications would have no benefit. For the remaining 6 applications, data layout may increase the overall array references that exhibit self or group-spatial locality. However, the benefit of data layout for these 6 applications, may be less than expected when the legality of data layout is considered.

5.3.3 Loop Permutation To Enhance Spatial locality

Section 5.3.1.1 evaluates the need for loop permutation at the level of individual array references. It does not consider conflicts among the array references belonging to the same nest. In contrast, this section reports on the need to use loop permutation to enhance spatial locality, when such conflicts are considered. The framework described in Section 3.7.3 is used to determine whether loop permutation needs to be applied to a given nest.

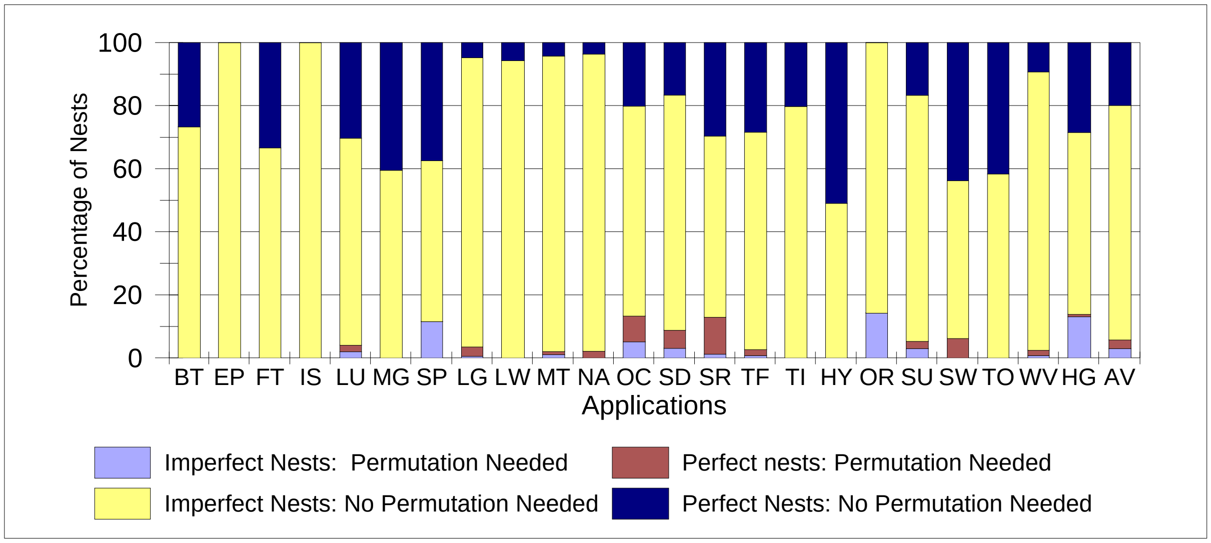


Figure 5.13: Percentage of nests requiring loop permutation for better spatial locality.

Figures 5.13 and 5.14 show the percentage of nests and contribution to execution time of nests, that may benefit from loop permutation, respectively. The nests are classified according to whether permutation is needed or permutation is not needed. Furthermore, the nests are classified as real perfect or not real perfect.

In two thirds (16) of the applications, loop permutation would have no benefits with respect to spatial locality. In particular, Figure 5.8 supports this finding for 13 of these 16 applications (BT, EP, IS, MG, LG, LW, MT, NA, HY, OR, SU, SW and TO). In fact, Figure 5.8 shows that for the 13 applications listed above, array references that would exhibit spatial locality due to loop permutation contribute to a very small percentage of execution time. For the remaining 3 of the 16 applications FT, SD and TI, innermost loop conflicts among array references in the same nest make loop permutation ineffective in making more array references exhibit spatial locality.

For the remaining 7 applications (LU, SP, OC, SR, TF, WV and HG), Figure 5.13 and Figure 5.14 show that loop permutation would increase the overall number and contribution to execution time of array references that exhibit spatial locality. These 7 applications warrant further consideration. The transformations discussed in the previous chapter are applied to the nests that require loop permutation and that are not perfect.

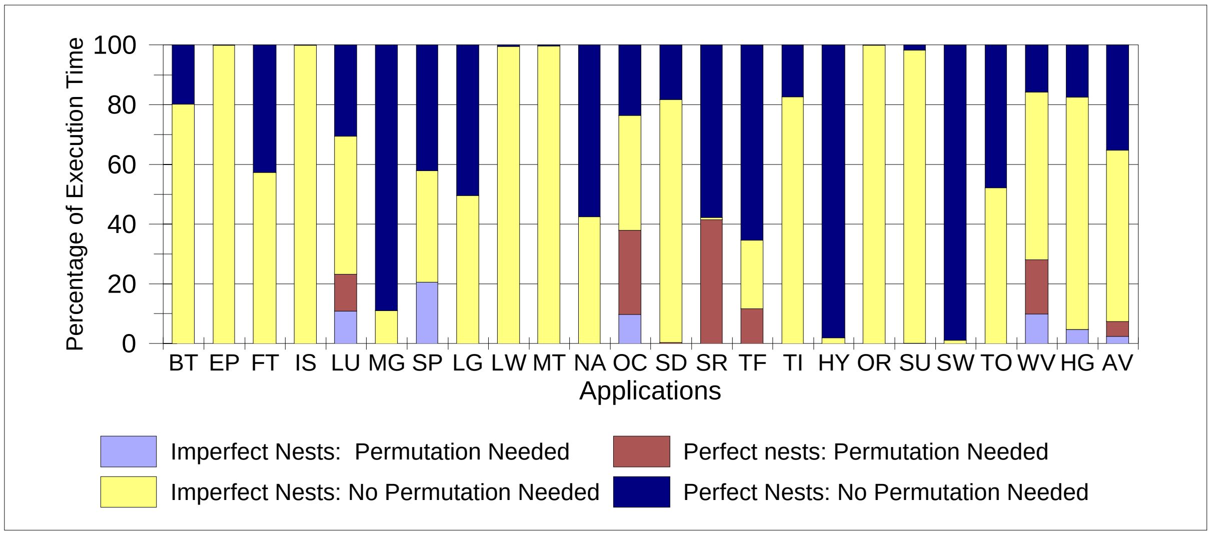


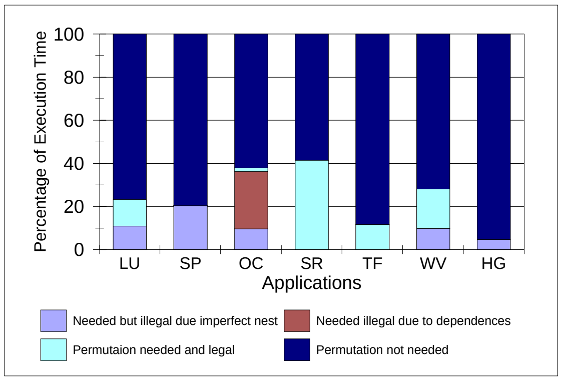
Figure 5.14: Percentage of execution time of nests requiring loop permutation for better spatial locality.

These transformations attempt to make loop permutation applicable by transforming the imperfect nests into perfect ones. Also the legality of loop permutation is examined for the nests that require loop permutation. The loop nests in the 7 applications are classified under four categories:

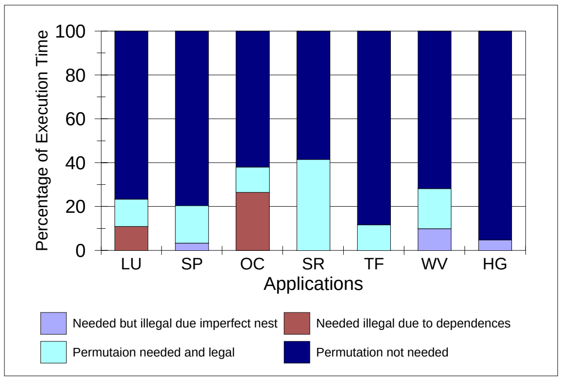
- Permutation is not needed.
- Permutation is needed and legal.
- Permutation is needed but is illegal due to dependences.
- Permutation is needed but is illegal because the nest is imperfect.

Figure 5.15 shows the classification of the loop nests in these 7 applications, before and after applying code sinking, loop distribution, and loop distribution with scalar expansion. The effects on these applications are discussed next.

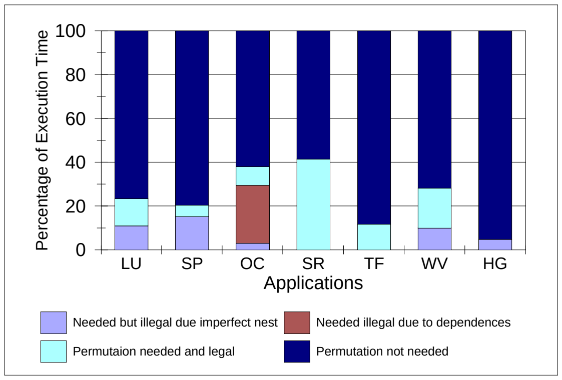
- For HG, all the nests that required loop permutation are imperfect. None of the transformations is able to transform these imperfect nests into perfect ones. This is consistent with the reuslts of the previous chapter that indicate that none of the transformations is effective for HG.



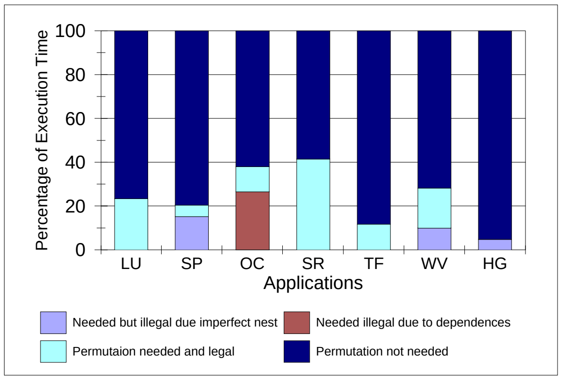
(a) Original



(b) Code Sinking



(c) Loop Distribution



(d) Dist. with Scalar Expansion

Figure 5.15: Effects of code sinking, loop distribution, and loop distribution with scalar expansion on the legality of applying loop permution when it is needed.

- For WV, originally, loop permutation is legal in most nests where it is needed. Some nests however are not perfect and permutation is not possible. None of the transformations is able to make these nests perfect.

- For SR and TF, loop permutation is legal in all the nests that required permutation.

- For OC, most nests that require loop permutation are perfect. However, for the majority of these nests loop permutation is illegal due to dependences. There are also some imperfect nests that require permutation. Code sinking and loop distribution with scalar expansion transform all these nests into perfect nests, where dependences allow the required permutation. Loop distribution is able to transform most (not all) of these nests into perfect nests, where the dependences allow the required permutation.

- For SP, all the nests that require loop permutation are imperfect. Code sinking transformed the majority of these nests into perfect nests where the required permutation is legal. In contrast, loop distribution— with or without scalar expansion — transformed only a few of these nests into perfect nest, where the dependences allow the required permutation.

- For LU, in half of the nests, the required permutation is legal. In the other half, loop permutation is illegal because these nests are not perfect. Code sinking transforms all these imperfect nests into perfect nests, however, the resulting dependences prevent the required permutation. In contrast, loop distribution with scalar expansion succeeds in transforming all the imperfect nests into perfect ones, where the dependences allow the needed permutation. Loop distribution has no effect on LU.

To summarize, the three transformations fail to enable permutation in 2 of the 7 applications. In 2 other applications, they are not needed. For the remaining 3 applications, the 3 transformations have different degrees of success in enabling loop permutation in imperfect nests. Therefore, and because of the overhead that these transformations introduce as seen in the previous chapter, loop distribution should be tried first; code sinking should be tried second; and finally, loop distribution with scalar expansion may be tried.

5.4 Summary

This chapter shows that perfect loop nests, that have at least a pair of permutable loops, contribute to less than 20%, in half of the benchmark applications. In general, increasing the relative contribution to execution time of perfect nests causes an increase in the relative contribution to execution time of permutable perfect nests. Loop distribution with scalar expansion achieves the highest relative contribution to execution time of permutable nests. It outperforms code sinking because distributing loops and replacing scalars with arrays simplifies the dependences in loop nests.

In only 6 applications, loop nests that may benefit from tiling contribute significantly to execution time. Code sinking and loop distribution with scalar expansion succeed in increasing the number of applications that may benefit from tiling to 11.

This chapter also shows that for 8 applications, the array references that can exhibit self-spatial locality due to loop permutation or a different data layout contribute significantly to execution time. Moreover, in 12 applications, changing the data layout can cause array references, with significant contribution to execution time, to exhibit group-spatial locality. On the one hand, however, when self and group-spatial locality are considered together, and data layout conflicts between references to the same array are considered, only 5 applications may benefit from flexible data layout. On the other hand, when inner loop conflicts, between array references in the same nest, are considered only 7 applications may benefit from loop permutation.

Chapter 6

Conclusion

This chapter concludes the thesis. First, it presents conclusions concerning loop nest structures. Second, it draws conclusions about cache locality. Finally, the chapter proposes future work.

6.1 Nest Structures

Many locality optimizations target the perfect nests of an application. This research shows that perfect nests, on average, contribute to only 39% of the execution time of the benchmarks application. Therefore, increasing the presence of perfect nests across the applications may make these optimizations more applicable, and hence, more beneficial.

Code sinking, loop distribution and loop fusion are three transformations that can transform an imperfect nest into a perfect one. These three transformations, along with loop distribution with scalar expansion, are implemented and applied to a suite of 23 applications.

This research finds that loop fusion is ineffective in transforming imperfect loop nests into perfect ones. In 9 applications, no transformation increased the relative contribution to execution time of perfect nests. However, for the 14 remaining applications, the average relative contribution to execution time of real perfect nests increased from 38% to 66% by code sinking, to 50% by loop distribution, and to 58% by loop distribution with scalar expansion. Loop distribution with scalar expansion causes the highest overhead, followed by code sinking and then by loop distribution which had negligible overhead. When applying code sinking, the size of the loop into which code is sunk should be considered carefully; when the size is small, the overhead may be too high. Similarly, when applying loop distribution with scalar expansion, the size of the arrays into which scalars are to be expanded should be small, otherwise the overhead becomes very high.

The assumption that loops account for more than 90% of the execution time of an application is not always true. For some applications, this assumption is false due to loops enclosing calls to routines with only straight-line code. In this case, in order for the optimizations targeting loops to be effective, inlining [Bal79, She77] or inter-procedural analysis [ASU86, FJ91] must be used.

6.2 Locality

Even though loop permutation is found legal for most perfect nests, the low contribution to execution time of perfect nests, makes loop permutation applicable in only a few applications. Code sinking and especially loop distribution with scalar expansion make loop permutation more applicable by increasing the relative contribution to execution time of perfect nests.

The potential benefit to temporal locality of tiling is found significant in only 6 applications. By transforming imperfect loop nests into perfect nests, code sinking and loop distribution with scalar expansion increase the potential benefit of tiling in 11 applications.

In most applications, most array references are found to exhibit self-spatial locality. This research shows that loop permutation may benefit spatial loacality in only 7 applications. For 3 other applications, loop permutation would be benefical to individual array references. However, due to inter-nest conflicts, loop permutation would have no benefit.

Changing the data layout of an array may make the references to this array exhibit self or group-spatial locality. This research shows that for 17 applications, data layout would have no benefit to spatial locality. This indicates that, generally, array layouts are carefully chosen by the application developers.

6.3 Future Work

Loop nests that require loop fusion to become perfect contribute significantly to the execution time of many applications. Neither loop distribution nor loop fusion are effective in transforming these nests into perfect nests. However, by sinking loops inside other loops, these loop nests may be made perfect. Hence, an extension to this work would be to allow code sinking of loops inside other loops. Also the shift-and-peel transformation [MA97] may be used to eliminate the dependences that prevent fusion from obtaining perfect nests.

The frameworks used to evaluate spatial locality should be unified in order to handle cases where both loop permutation and data layout are needed. Also the framework can be extended to consider parallelism requirements. When parallelism is not considered, loop permutation and data layout are often not needed because most array references exhibit spatial locality already. However, to satisfy parallelism requirements and maintain good spatial locality the need for loop permutation and data layout may become greater. The framework for data layout can also be extended to handle the aliasing of arrays, and to propagate the effects of changing the layout of arrays across procedures.

Appendix A

System Detail

This appendix describes in greater detail the approach used in this thesis. In particular, it presents an implementational view of the system shown in Figure 3.2. First, the new passes added to Polaris are described. Second, the F77 system is presented. Third, the information held in the database is listed. Finally, the post-processing units are discussed.

A.1 New Polaris Passes

During the course of this research several passes have been added to the Polaris passes. Each of these passes can be classified under one of the following categories:

- The *transformation* pass implements a specific transformation (e.g. loop fusion).
- The *instrumentation* pass adds routine calls to provide run-time information (e.g. size of arrays allocated by *malloc*).
- The *profiling* pass gathers various application metrics and characteristics (e.g. number of loops).

Table A.1 shows and describes all the passes added to Polaris in this research. These passes implement the transformations applied, add instrumentation calls to provide the run-time information, and gather profile information about the benchmark applications. Polaris provides a "switches" file to control which passes are applied when the compiler is processing a given application.

A switch called "Transform" controls which transformation to apply, if any. Code sinking, loop distribution, loop distribution with scalar expansion, and loop fusion can

Pass Type	Pass Name	Description
Transformation	code sinking	applies code sinking
	loop distribution	applies loop distribution
	scalar expansion	applies loop distribution with scalar expansion
	loop fusion	applies loop fusion
	inlining	inlines specific program units
Instrumentation	loop timing	inserts loop timing calls
	routine timing	inserts routine timing calls
	routine counting	inserts routine counting calls
	array size	inserts calls to output the array sizes used in scalar expansion
Profiling	loop profiling	extracts inforamtion about loops
	program profiling	extracts information about program units
	array profiling	extracts information about array references

Table A.1: Passes added to Polaris

be applied by setting “Transform” to 1, 2, 3, and 4, respectively. If “Transform” is set to 0 then no transformation is applied. To apply the inlining transformation, the switch “AddInliningAssertions” should be set to 1. The timing of loops and program units and the counting of how many times each program is called are controlled by the switch “InsertTiming”. The calls to print the array sizes allocated by “Malloc” are inserted by setting the “ProfileMalloc” switch to 1. To extract information about loops and programs units, the switch “Profile” is set to 1. To extract information about the array references, the switch “Arrref” is set to 1.

A.2 The F77 System

The F77 system represents a system that compiles and executes Fortran-77 programs. This research uses the Silicon Graphics Incorporated’s Fortran-77 compiler. This compiler provides mechanisms for dynamic memory allocation, use and deallocation, which are not available in other Fortran compilers (e.g. g77). Programs are compiled using the default optimization setting (i.e. option -O1). Executables produced by this compiler are run on an SGI Challenge. The Challenge runs IRIX 5.3, has eight MIPS R4400 processors and has 512 megabytes of RAM.

A.3 Database

The database consists of files containing the profiling information. Program and loop information are combined with timing information and are stored in files with ".info" extension. The information about array references is stored in files with ".ref" extension. The following list describes the main elements of the information kept in the database for every loop.

1. The loop identifier.
2. The statement numbers of the header and the tail.
3. The level, i.e. the number of loops enclosing this loop.
4. The maximum level of all the loops enclosed by this loop.
5. The bounds and step.
6. The net and total times.
7. Belongs to a perfect nest or not.
8. Safe or not.
9. A list of all the loops directly enclosed by this loop.
10. A list of all the loops enclosing this loop.
11. A list of all the loops with which this loop can be permuted.
12. A list of all the subroutines directly enclosed by this loop.
13. A list of the array references directly enclosed by this loop.

The following list describes the main elements of the information kept in the database for every array reference.

1. The name of the array.
2. The dimension of the array reference.
3. The variable dimension of the array reference.
4. The contribution to execution time of the reference.
5. A list of all loops enclosing the array reference.
6. A list of the subscript expressions of the array reference.
7. Preferred loop ordering for self-spatial locality.
8. Preferred loop ordering for self-temporal locality.

Pass Name	**Description**
printloops	prints the loops information
perfect	computes the number and the execution time of perfect nests
perfdim	computes the number of perfect nests with different dimensions
loopstruct	computes the execution time of different nest structures
permutable	computes the number and the execution time of permutable perfect nests
tiling	computes the potential benefit of tiling
printarref	prints the array references information
arrref	classifies the array references into different categories
datalayout	determines whether data layout is needed
loopperm	determines whether loop permutation is needed

Table A.2: Post-processing passes.

A.4 Post-Processing

The post-processing units use the information stored in the database in order to derive more complex results (e.g. relative contribution to execution time of perfect nests). These post-processing units are independent of Polaris. However, similar to the Polaris switches file, a switch file (called "pswitches") is used to select which post-processing units to run. In this research, several post-processing passes are implemented and are listed in Table A.2. These passes reflect the results presented throughout this thesis.

Bibliography

[AK84] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 233–246, Montreal, Quebec, Canada, June 1984.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Bal79] J. E. Ball. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 214–220, Denver, Colorado, August 1979.

[BBB$^+$91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, and R. L. Carter. The NAS parallel benchmarks. *Int. J. Supercomp. Appl.*, 5(3):63–73, 1991.

[BCP$^+$89] M. Berry, D. Chen, P.Koss, S. Lo, and Y. Pang. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, CSRD, University of Illinois, May 1989.

[BDE$^+$96] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[BEF$^+$95] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of the 7th Workshop on languages and Compilers for Parallel Computing*, pages 141–154, Springer-Verlag, Berlin, 1995.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6^{th} International Conference on Architectural Support for Programming languages and Operating Systems*, pages 252–262, San Jose, CA, October 1994.

[Dix92] K. M. Dixit. New CPU benchmarks from SPEC. *Digest of Papers, Spring COMPCON 1992, Thirty-Seventh IEEE Computer Society International Conference*, pages 305–310, February 1992.

[FJ91] Charles N. Fischer and Richar J. LeBlanc Jr. *Crafting a Compiler with C.* The Benjamin/Cummings Publishing Company, 390 Bridge Parkway Redwood City, CA 94065, 1991.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Ontario, Canada, June 1991.

[IT88] F. Irigoin and R. Tiolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, USA, January 1988.

[KM92] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, Washington, DC, July 1992.

[KRC97] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.

[Kuc77] D. J. Kuck. A survey of parallel machine organization and programming. *Computing Surveys*, 9(1):29–59, March 1977.

[Kuh80] R. Kuhn. *Optimization and interconnection complexity for: parallel processors, single-stage networks, and decision trees.* PhD thesis, University of Illinois at Urbana-Champaign, February 1980.

[LRW91] M. Lam, E. Rotherberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4^{th} International Conference on Architectural Support for Programming languages and Operating Systems*, Santa Clara, CA, April 1991.

[MA95] Naraig Manjikian and Tarek S. Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8^{th} International Conference on Parallel and Distributed Computing Systems*, pages 111–118, Orlando, FL, September 1995.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[Man97] Naraig Manjikian. *Program transformations for cache locality enhancements on shared-memory multiprocessors.* PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1997.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, Toronto, Ontario, Canada, June 1991.

[MT96] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Proceedings of the 7^{th} International Conference on Architectural Support for Programming languages and Operating Systems*, pages 94–104, Cambridge, MA, October 1996.

[MW96] Joseph Michael Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, 390 Bridge Parkway Redwood City, CA 94065, 1996.

[NCS] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. *Information on their scientific applications is available at the website http://zeus.ncsa.uiuc.edu:8080/archives/.*

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 8(12):1184–1201, December 1986.

[She77] R. W. Sheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977.

[WL91a] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[WL91b] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.