C-to-C Cache Based Optimizations

Master’s Thesis submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Embedded Systems Design

presented by
María Haydeé Rodríguez Blanco

under the supervision of
Prof. Mariagiovanna Sami
co-supervised by
Prof. Dr. rer. nat Rainer Leupers,
Dipl-Ing. Felix Engel

September 2012
I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

María Haydeé Rodríguez Blanco
Lugano, 14th September 2012
To my beloved parents Ulises and Flor,
and my brothers Juan and Pedro.
“Together forever and never apart,
maybe in distance, but never in heart.”
Anonymous
Abstract

The increasing demands on performance and cost for embedded systems design, makes necessary to look for the development of efficient software that leverage the capabilities of the modern architectures. In order to meet the performance constraints of a specific design, it is possible to perform a customization of the source code for the target compiler. Modern embedded compilers differ in their capabilities of optimization and they can also perform different under different coding styles. Therefore, to manually tune the source code is a process that not only requires time, but also certain expertise. Knowledge about the architecture and the optimizations that can give greater benefits is required.

A source-to-source compiler is a tool that can perform automatically the task of optimizing the source code that serves as input to the target compiler. Some benefits given by the source-to-source compiler is that the tool can be implemented as a re-targetable tool, i.e. it can be reuse for several architectures. The decisions regarding the application of a set of optimizations such as their configuration, order and profitability are taken externally to the target compiler, then benefits over the ones obtained by the target compiler can be achieved. Finally, optimizations that are not included in the target compiler can be applied at source level. This work is focused in the exploration of code transformation that can be placed in a C-to-C compiler.

In the past, applications related with embedded systems assumed small data sets and therefore the capacity of the cache memory were enough. Thus, embedded compilers have not exploited optimizations to improve the cache usage as it has been done in high-performance computing. However, the increasing demand of computing power and the increasing size of the data sets make the cache efficiency to become relevant, which raises the interest to look at the cache based optimizations as promising source-to-source transformations. In this thesis, it is explored specifically the loop reordering transformations, a subset of the possible cache based optimizations that only reorders the iterations of the loops without deleting or adding any execution of any statement.

The work makes two main contributions the characterization of the a set of loop reordering transformations so that a theoretical basis is given for future developments in the automation of the application process, and a case study was also performed to analyze its applicability as C-to-C transformations for real-world applications. From the first objective, one of the main conclusions is that improvement in performance were seen for some architectures, but some target embedded compilers like the one of Tensilica Diamond 570 already includes some of them as part of its optimization phase, which means there is no space for the application at source level. The characterization also shows that its profitability depends on different program context factors and architectural factors. By the other hand, in the case study, it is shown that the analysis of the legality of the transformation in order to preserve the correctness of the code can be a limiting factor for the applicability in real world applications.
Acknowledgements

This thesis was developed at the Institute Software for Systems on Silicon (SSS) of RWTH Aachen University, Germany. I would like to thank my supervisor Prof. Rainer Leupers for providing me with the opportunity to perform my thesis with his group. I express my sincere thanks to my tutor Felix Engel for his support and guidance throughout all my work. I would like to thank also to all the assistants in SSS for the nice working environment they offered me. At last, I am very grateful to Dominik Auras not only because of the time he invested in carefully proof-reading this work, but also because he has truly lived and shared with me all my efforts for reaching the end. His encouragement will never be forgotten.

Then, I also want to sincerely thank to the ALaRI committee, especially to Ing. Umberto Bondi and Professor Mariagiovanna Sami, for granting me a scholarship and giving me the opportunity to be part of the master program that has certainly enriched my professional career. Moreover, I wish to thank to all the ALaRI members and colleagues for every day spent together. Additionally, this work and my studies were financially supported by the MICIT and CONICIT of the Government of the Republic of Costa Rica, to whom I am particularly indebted.

Above all, I thank God for his gift of faith, family and friends. Great thanks to my forever friend Cristina, who had shared with me everything during this journey. My new brother Rafael, from whom I had learned a lot. To Miguel, my light and support since the start to the end. I cannot forget David and Pablo who, although the distance, were always there.

Finally, my biggest thanks go to my parents Ulises and Flor without whom I would not have achieved my dreams. I owe my parents much of what I have become. I dedicate this work to them, and to my beloved brothers Juan and Pedro, to honor their love, patience, and support throughout my entire studies.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Organization of the Thesis</td>
<td>3</td>
</tr>
<tr>
<td>2 Fundamentals of Loop Reordering Transformations</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Memory Hierarchies</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Caches</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Data Locality</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Terminology for Loop Reordering Transformations</td>
<td>7</td>
</tr>
<tr>
<td>2.4.1 Data dependences</td>
<td>7</td>
</tr>
<tr>
<td>2.4.2 Data Dependence Graph</td>
<td>7</td>
</tr>
<tr>
<td>2.4.3 Loops and Nested Loops</td>
<td>8</td>
</tr>
<tr>
<td>2.4.4 Dependences in Loops</td>
<td>11</td>
</tr>
<tr>
<td>2.4.5 Distance and Direction Vectors</td>
<td>12</td>
</tr>
<tr>
<td>2.4.6 Direction Matrix</td>
<td>13</td>
</tr>
<tr>
<td>2.4.7 Loop-Carried and Loop-Independent Dependences</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Data Reuse Analysis</td>
<td>14</td>
</tr>
<tr>
<td>2.6 The CoSy Compiler Development System</td>
<td>15</td>
</tr>
<tr>
<td>2.6.1 Loop Information in CoSy</td>
<td>16</td>
</tr>
<tr>
<td>2.6.2 CoSy Engines</td>
<td>19</td>
</tr>
<tr>
<td>2.7 Related Work</td>
<td>20</td>
</tr>
<tr>
<td>2.7.1 Loop Cost Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>3 Evaluation of Loop Reordering Transformations as C-to-C Transformations</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Simulation Environment: Memory Characteristics and Tools</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Loop Permutation</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1 Definition</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2 Legality Test</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3 Profitability of Loop Permutation</td>
<td>25</td>
</tr>
<tr>
<td>3.2.4 Source Code for the Evaluation of Loop Permutation</td>
<td>31</td>
</tr>
</tbody>
</table>
3.2.5 Results ................................................................. 32
3.2.6 Conclusions for Loop Permutation ............................. 40

3.3 Loop Reversal .......................................................... 40
3.3.1 Definition ............................................................. 40
3.3.2 Legality Test ......................................................... 41
3.3.3 Profitability of Reversal ........................................... 41
3.3.4 Source Code for the Evaluation of Reversal ................. 42
3.3.5 Results ................................................................. 42
3.3.6 Conclusions for Loop Reversal .................................. 45

3.4 Loop Skewing ............................................................ 45
3.4.1 Definition ............................................................. 45
3.4.2 Legality Test ......................................................... 46
3.4.3 Profitability of Skewing ........................................... 46
3.4.4 Source Code for the Evaluation of Skewing ................. 48
3.4.5 Results ................................................................. 49
3.4.6 Conclusions for Loop Skewing .................................. 51

3.5 Tiling ................................................................. 52
3.5.1 Definition ............................................................. 52
3.5.2 Legality Test ......................................................... 53
3.5.3 Profitability of Tiling .............................................. 53
3.5.4 Source Code for the Evaluation of Tiling ................. 54
3.5.5 Results ................................................................. 54
3.5.6 Conclusions for Loop Tiling .................................. 57

3.6 Summary ............................................................... 57

4 Case Study: C-to-C Compiler Extended for Loop Reversal ................................. 61
4.1 Application of Loop Reversal ........................................ 61
  4.1.1 Loop Reversal Example ........................................... 62

4.2 Implementation Details: Loop Reversal .......................... 63
  4.2.1 Loop Requirements ............................................... 63
  4.2.2 Program Flow .................................................... 65
  4.2.3 Exploration of the Data Dependence Graph .............. 67
  4.2.4 Additional Considerations ..................................... 68

4.3 Integration into a CoSy Compiler ................................ 69

4.4 Tests Setup ........................................................... 70
  4.4.1 Results ............................................................. 70

4.5 Summary ............................................................... 71

5 Conclusions and Outlook ............................................. 73

Bibliography ............................................................. 75
Figures

2.2 Example of a data dependence graph [14] ............................... 8
2.3 Example of a loop at the Control-Flow Graph level ................... 9
2.4 Two dimensional iteration space .......................................... 11
2.5 Example of loop markers tree structure .................................. 19

3.1 Loop carried dependences on 3-way nesting sample code code ........ 32
3.2 Speedup when permuting on (a) 3-way nesting and (b) 2-way nesting .... 33
3.3 Comparison between original(a) and transformed(b) assembly code for N = 300 .... 35
3.4 Speedup when applying (a) only reversal and (b) reversal and permutation ... 43
3.5 Effect of Skewing on the iteration space: (a) original and (b) transformed .... 46
3.6 Speedup when applying (a) only skewing and (b) skewing and permutation ... 50
3.7 Speedup when applying tiling on the ARM926EJ-S .......................... 54
3.8 Speedup when applying tiling on the TI-C6455 ........................... 56
3.9 Speedup when applying tiling on the Tensilica Diamond 570 ............... 57

4.1 Iteration range of the Loop Case 1 (a) original and (b) transformed ....... 63
4.2 Checks for each loop when traversing the loop marker tree in post-order .... 66
4.3 Check for data dependence information flow diagram ........................ 67
Tables

2.1 Kinds of data dependences .................................................. 8
2.2 Locality cost in terms of cache lines ................................. 22

3.1 Memory characteristics used for the cycle accurate simulators .................................................. 24
3.2 Loop Cost to derive the “best” loop order ................................................. 28
3.3 Stride analysis when applying loop permutation on 2-way nesting code .............................................. 34
3.4 Stride analysis when applying loop permutation on 3-way nesting code .............................................. 34
3.5 Loop Cost to derive the “best” loop order on 3-way nesting code .................................................. 36
3.6 Loop Cost to derive the “best” loop order on 2-way nesting code .................................................. 36
3.7 Break-even-point where problem size exceeds cache capacity in 3-way nesting code .................................. 37
3.8 Break-even-point where problem size exceeds cache capacity in 2-way nesting code .................................. 37
3.9 Software pipelined scheduling in TI-C6455 when applying loop permutation .............................................. 38
3.10 Cache statistics in 3-way nesting when applying loop permutation .................................................. 39
3.11 Stride analysis when applying loop permutation on reversal sample code .............................................. 44
3.12 Loop Cost to derive the “best” loop order on reversal sample code .................................................. 44
3.13 Break-even-point where problem size exceed cache capacity .......................................................... 45
3.14 Software pipelined scheduling when applying skewing and permutation .............................................. 51
3.15 Summary of the data reuse in each loop for the matrix multiplication code .............................................. 54
3.16 Cache misses exposed in each loop for the matrix multiplication code .............................................. 55
3.17 Cache performance in TI-C6455 when applying loop tiling .......................................................... 56
3.18 Summary of factors influencing the profitability .......................................................... 59

4.1 Summary of loop reversal .................................................. 63
4.2 Implications for level information annotated in the mirlcdEdge structure .............................................. 68
4.3 Results of applying loop reversal automatically in the benchmark suite .............................................. 70
4.4 Statistics on applications where data dependence graphs were found .............................................. 71
4.5 Speedup obtained where loop reversal was applied .......................................................... 71
Chapter 1

Introduction

1.1 Motivation

The demands on performance and cost for embedded system designs are always increasing. The presence of efficient software is mandatory to fully realize the capabilities of today’s architectures. When creating software, programmers are primarily concerned about programming-style and structuring, in order to keep clarity for ease of debugging, verification and maintenance, and only then follows efficiency [16]. Programming at a high abstraction level facilitates the production of well-structured and easy-to-read code. However, experience shows that if the level of abstraction is higher, then the code is less hardware specific. This can lead to inefficiencies when executing the code and in the final product efficiency is what matters.

Therefore, to achieve the performance goals required in a system, compilers implement several program transformations that leverage the potential of the target. Architectural features to be exploited include for example: multilevel memory hierarchies, scalar or SIMD registers, prefetching mechanisms and pipeline architecture. The definition of how, when and which transformations should be applied, is one of the biggest challenges when designing a compiler, not only due to the high number of possible transformations, but also due to the complex interaction between them. Some of the problems found in the process of applying a transformation are:

- A transformation may degrade performance. If it generates a profit must be determined.
- The order in which a set of transformations is applied affects the resulting performance.
- The influence of each code transformation depends on its configuration that when applied with different values leads to variations on the profit.

A common solution used by modern compilers is to use internal heuristics to choose a set of transformations. These heuristics are target-specific, and in most of the cases involve fixed criteria or fixed configurations, which is not always precise and may result in degrading the performance. Consequently, the generated code might not be as optimized as it could be by manually tuning the code.

The first stage of a compiler, known as front-end, lowers the input program towards the first intermediate representation (IR). Then, the first optimization phase (the high-level optimizations) is applied. The high-level intermediate language is semantically the same to the original
source program \[5\]. Therefore, most of these high-level optimizations can also be explicitly performed on the original source code. By doing so, an improved source code is produced, which serves as new input for the compiler. The optimizations applied directly in the source code are called, hereafter, *source-to-source transformations*. The application can be done automatically by means of a source-to-source compiler. The application of these transformations by means of a source-to-source compiler has several benefits:

- The process of applying a transformation at source level can be placed on top of the common flow of the software development process. In the common flow the source code is the input to the target compiler, who converts this code to a target specific binary code. But now a new element can be place before, the source-to-source compiler, who generates a new and improved input to the specific target compiler. Note, this is a single tool that can be reused for different targets.

- It externalizes the decision regarding the application of an optimization. Thus, how, when and which transformations to apply can be decided by an external agent that might have a better knowledge or more information than the one obtained by the compiler, so that the resultant performance is better.

- Optimizations that are not included in a target compiler, can be applied at source level.

The application of a source to source transformation suffers from the same problems that a normal compiler encounters, for example, the determination of its profitability, configuration and order, which furthermore are target specific. However, it merits to spend a greater effort in solving these problems at a source level due to the reusability offered.

In the recent past, caches were not commonly considered in embedded systems software design. Memories were assumed to be small and to have fixed access times. Consequently, the compilers have not taken care about the cache related improvements. Typical transformations for this purpose have been left aside. However, due to the increasing demand of computing power and the increasing size of the data set handled by embedded systems, the cache efficiency become relevant. This raises the interest to look at the cache-based optimizations as promising source-to-source transformations.

In general, cache performance is improved by reducing the time between uses of the same data. This is known as enhancement of locality and it can be achieved by for example loop re-ordering transformations. Loop reordering transformations are a subset of the possible cache-based optimizations. They only reorder the iteration of the loops without deleting or adding any execution of any statement. These optimizations in particular, are well-known transformations, whose effects has been subject of study for high-performance computers. According to its definition, the nature of these transformations is manipulate the code only, which make them an ideal candidate as source-to-source transformations.

This thesis makes the following contributions:

1. It performs an evaluation of loop reordering transformations as source to source transformations in order to determine if improvements seen for high-performance computing systems are also obtained in embedded systems. The evaluation also characterizes the transformations so that a theoretical basis is given for future developments in the automation of the application process of these transformations.

2. It implements a selected transformation in an automatic transformation tool, as a case study to analyze its applicability for real-world applications.
1.2 Organization of the Thesis

The remainder of this document is organized as follows.

- **Chapter 2** presents the terminology related to loop transformations and describes CoSy, the framework used to implement a selected loop transformation.

- In **Chapter 3**, the analysis done for evaluation of loop reordering transformations as C-to-C transformations is presented.

- **Chapter 4** describes a proof-of-concept automation of one selected transformation.

- Finally, in **Chapter 5**, directions for future work and conclusions are drawn.
Chapter 2

Fundamentals of Loop Reordering Transformations

This chapter provides the background of loop reordering transformations as a technique of code optimizations for the improvement of cache performance. Memory hierarchies, caches memories and data locality are first introduced. Then, the chapter sets the terminology that is used along the document regarding the transformations. In Chapter 4, a proof-of-concept implementation of one selected transformation using the CoSy compiler framework is presented. Therefore, a brief description about the main characteristics of CoSy is presented. The chapter concludes with a review of the related work on loop reordering transformations.

2.1 Memory Hierarchies

Memory hierarchy is the term given to the organization of the memory subsystem where the most frequently used data are placed closer to the CPU. Because the smaller and expensive memories are more faster, they are placed closer to the CPU. When moving away the slower, but also larger and less expensive memories are found. The concept is illustrated in Figure 2.1, which also gives typical numbers for speed and capacity of cache in embedded, desktop and

Figure 2.1. Memory Hierarchy [11].
Looking at the difference in speed between the cache and the main memory, it is clear why the improvement of the cache performance is important. Whenever a variable needs to be loaded from memory it is searched for first in the cache. If it is found there, then the load delay is approximately 1 cycle. Otherwise, it needs to be retrieved from main memory, which implies that the load delay can reach 100 cycles. Consequently, if it is possible to improve the probability that a variable is in cache whenever it is needed, then great improvements can be achieved in the overall performance.

2.2 Caches

Cache memories are small and fast storage devices used to improve the average access time of slow memories. When the processor finds a required data element in cache, this is called a cache hit. If the processor does not find the needed datum in cache then a miss occurs and it must be fetched from main memory. The movement of data is performed in basic units called cache lines (or blocks). The blocks consist of multiple words for efficiency reasons [11].

The decision of where to place a block in cache is determined depending on the set associative scheme used. Set refers to a group of blocks in cache. A mapping of the block address to the set needs to be performed in order to find a block. The set is chosen by the address of the data as follows:

\[(\text{block address}) \mod (\text{Number of sets in cache})\] (2.1)

If there are \(n\) blocks in a set, the cache layout is called: \(n\)-way set associative. A direct-mapped cache, instead, has just one block per set, which means that a block is always placed in the same location. And a fully-associative cache has just one set, so a block can be placed anywhere. The benefits of different cache organizations depend on the miss rate. Miss rate is the fraction of cache accesses that result in a miss. The causes of high miss rates can be better analyzed by classifying the misses in:

- Compulsory: the first access to a block is not in the cache, so the block must be brought into the cache. They occur even if we have infinite cache.
- Capacity: if the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later reloaded.
- Conflict: a block can be discarded and later retrieved if too many blocks map to the same set. They occur only if the placement strategy is not fully-associative [11].

2.3 Data Locality

A program has a good data locality in memory if it accesses often the same data it has recently used. There are two notions of data locality: temporal locality and spatial locality. Temporal locality occurs when the same data element is used several times within a short time period. Spatial locality occurs when different data elements that are located near to each other are also used within a short period of time [1]. Listings 2.1 and 2.2 show an example where data elements expose data locality.
2.4 Terminology for Loop Reordering Transformations

Loop reordering transformations reorder the computation performed in loop nests, and might increase the probability of keeping data in the cache between uses. They change the code without adding or deleting any execution of any statement. Examples of those locality-enhancing loop transformations are permutation, tiling, reversal, skewing, distribution, fusion, etc.

For the evaluation of loop reordering transformations done in this thesis, some terminology is needed. The following subsections present definitions and explanations of first, the general concept of data dependences and its representation in a graph, then an introduction to the fundamentals of loop and loop nest which gives the basis for the definition of dependences specifically in loops. Related to loop dependences, the concepts of distance vector, direction vector and direction matrix are presented and finally a classification on loop-carried and loop independent dependences is also described. After the definition of the terminology, there is also an explanation of data reuse.

2.4.1 Data dependences

A data dependence can be defined as follows:

“A statement \( S_2 \) depends on a statement \( S_1 \), if and only if: both statements access the same memory location and at least one of them stores into it, also there must be a feasible run-time execution path from \( S_1 \) to \( S_2 \).” [14]

The data dependences establish a set of constraints that ensure that any reorder that changes them is not allowed. Then, by analyzing the dependences of the program it is possible to preserve the correctness.

There are three kinds of data dependences: true dependences also known as Read-After-Write dependence (RAW), anti-dependences or Write-After-Read (WAR) dependences and output dependences or Write-After-Write (WAW). Their characteristics are summarized in Table 2.1.

2.4.2 Data Dependence Graph

The data dependence graph (DDG) is an abstraction used to represent the dependences of a program. It is a directed graph \( G = (V,E) \), having a set of nodes \( V = \{v_1,v_2,...,v_n\} \) that represent the assignment statements of the program \( \{S_1,S_2,...,S_n\} \), and the set of directed edges
## 2.4 Terminology for Loop Reordering Transformations

### Table 2.1. Kinds of data dependences

<table>
<thead>
<tr>
<th>Definition</th>
<th>Anti-dependence</th>
<th>Output dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>True dependence</td>
<td>The first statement reads from a location into which the second statement later stores.</td>
<td>Ensures that the second statement receives the value computed by the first one.</td>
</tr>
<tr>
<td>Anti-dependence</td>
<td>The first statement writes into a location that is later read by the second statement.</td>
<td>It prevents the transformation from introducing a new true dependence that did not exist in the first place by incorrectly interchanging the statements.</td>
</tr>
<tr>
<td>Output dependence</td>
<td>Both statements write to the same location.</td>
<td>This dependence prevents an interchange that might cause a later statement to read the wrong value.</td>
</tr>
</tbody>
</table>

**Notation**

\[ E = \{e_1, e_2, ..., e_x\} \]

represents the data dependence constraints among the statements. Therefore, if there exists a data dependence from statement \( S_1 \) to statement \( S_2 \), then there is an edge \( e = (v_1, v_2) \in E \), where \( v_1 \) and \( v_2 \) are respectively, the source \( (S_1) \) and the sink \( (S_2) \) of the data dependence. The data dependence graph for the example in Listing 2.3 is shown in Figure 2.2.

![Figure 2.2. Example of a data dependence graph](image)

**Listing 2.3. Vector swap example**

```
t = a[i]; /* S1 */
a[i] = b[i]; /* S2 */
b[i] = t; /* S3 */
```

### 2.4.3 Loops and Nested Loops

A loop can be defined at the programming level or at a control-flow graph (CFG) level. A loop at programming level is defined as a set of instructions that is repeated until a condition is
Terminology for Loop Reordering Transformations

A CFG-loop, instead, is a set of basic blocks $S$ with the following properties:

1. From any basic block in $S$ there is a path of directed edges that leads to a header basic block $h$.
2. There is a path of directed edges from $h$ to any basic block in $S$;
3. There is no edge from any reachable basic block outside $S$ to any basic block in $S$ other than $h$.

An illustration of a CFG loop is depicted in Figure 2.3. A basic block that is inside the loop and has a predecessor that is outside the loop is called the loop entry basic block. Additionally, a basic block that is inside the loop and has as successor a basic block that is outside the loop is called a leave basic block. The successor is called loop exit block. Note that a loop may contain multiple leave and loop exit basic blocks.

A loop element is defined as a sub-graph of the control-flow graph (CFG), that has an specific role. The following are four loop elements that can be identified within a loop:

- **Init**: the statements where the loop variables are initialized. *Loop variables* refers to those variables that are updated in every iteration of the loop.
- **Test**: the expression that sets the condition to decide whether the loop continues or exits.
- **Body**: the statements constituting the loop body
- **Incr**: the statements related with the updates of the loop variable.
The statements involved in each specific role can be contained in one or more basic blocks. Also a CFG-loop can be identified according to the organization of the loop elements in while-do and repeat-until structures. The main difference between them, is the possible exits of the loop and the loop element that correspond to the header \([4]\).

In this work specifically a variant of the while-do loop, the for-do loops are considered. The for-do loop starts with initialization statements which are used for the initialization of the loop variables. After the loop body statements have been executed the increment statements are executed which are used for updating the loop variables. How for-do loop looks like is shown in Listing 2.4.

Listing 2.4. for-do loop pseudo-code \([4]\)

\[
\begin{array}{l}
\text{initialization statements} \\
\text{for (test-expression) do} \\
\text{\quad \{} \\
\text{\quad \quad loop body statements} \\
\text{\quad \quad increment statements} \\
\text{\quad \}}
\end{array}
\]

In the programming language C, this loop structure is called for, in FORTRAN instead is called do. Now, the C-for-loop (hereafter simply loop) at programming level can be analyzed.

A for-loop is a structured program construct consisting of a loop header and a loop body.

Listing 2.5. for-loop example

\[
\text{for(i = b0; i <= bf; i += s) \quad \Rightarrow \text{header}} \\
\text{<body>i};
\]

An example of a for-loop is given in Listing 2.5 where \(i\) is the loop index variable that takes on values beginning at the lower bound \(b0\) in steps of size \(s\) until the upper bound \(bf\) is exceeded, and each value represents one loop iteration. The loop body contains statements in which the variable \(i\) may appear.

A loop nest \(L\) is a set of loops and their respective bodies structured such that exactly one loop \(l_{outer} \in L\) encloses all of the remaining loops, and no enclosing loop uses the same index variable as one of the loops it encloses. The nesting level of a loop is the number of loops which enclose it. Therefore, by definition the level of \(l_{outer}\) is 0, and the rest of the loops are numbered starting from the outermost to innermost. The depth of the loop nest is one larger than the maximum level of any loop that is part of the loop nest, i.e., \(depth = (max_{l \in L} level(l)) + 1\) \([17]\). An example of a 2-depth loop nest and the corresponding nesting levels is shown in Listing 2.6.

Listing 2.6. Two-depth loop nest

\[
\begin{array}{l}
\text{for(i=1; i<=4; i++) \quad \Rightarrow \text{Level 0}} \\
\text{for (j=1; j<=4; j++) \quad \Rightarrow \text{Level 1}} \\
\text{<body>};
\end{array}
\]
2.4 Terminology for Loop Reordering Transformations

2.4.4 Dependences in Loops

To describe precisely the dependences in loops, it is necessary to parametrize the statements by some representation of the loop iterations in which the statement occurs. To do so, three new concepts need to be added: the iteration number, the iteration space and the iteration vector.

Every value of a loop index has associated an iteration number, which usually correspond to the same value of the loop index. A normalized version of the iteration number can be defined, and it runs from 1 to the upper bound, in increments of 1. For calculating it the following formula can be used:

\[ \text{Iteration Number} = \frac{I - L + S}{S} \]  \hspace{1cm} (2.2)

where \( I \) is the value of the loop index on that iteration, and the loop index of the loop runs from \( L \) to \( U \) in steps of \( S \).

In a loop nest of depth \( m \), the loop bounds define a set of points in a \( m \)-dimensional discrete space called iteration space. Figure 2.4 shows an example of the iteration space of the nested loop shown in Listing 2.6. The solid line represents the way the iteration space is traversed with the loop nest configuration of the code. Each point contained in this space represents a possible iteration of the loop nest, which can also be denoted by using an iteration vector. Formally defined as a vector of integers that represents the iteration numbers for each of the loops in order of the nesting level \([14]\). In other words, it is defined as

\[ i = (i_0, i_1, \ldots, i_n) \]  \hspace{1cm} (2.3)

where \( i_k \), with \( 0 \leq k \leq n \), represents the iteration number for the loop at nesting level \( k \).

Then the iteration vector is used to parametrize the statements in the loop and in this way a more precise description of the loop dependence is achieved. For example, \( S[(2,1)] \), represents the instance of the statement \( S \) that occurs on the second iteration of \( i \) and the first iteration of \( j \), being \( j \) the innermost loop in a 2-nested loop.

It is important to determine the order in which the statements involved occur in order to accurately describe a data dependence. To do so, in loop dependences a lexicographic order is defined for the iteration vectors as follows:

"On iteration vectors of length \( n \), iteration \( i \) precedes iteration \( j \), denoted \( i < j \), if and only if\[i = (i_0, i_1, \ldots, i_n) \] and \( \forall k \leq n, i_k \leq j_k \]."
1. \( i[1:n-1] < j[1:n-1] \), where \(<\) denotes again the precedence of the subvector \(i[1:n-1]\) over the subvector \(j[1:n-1]\)
2. \( i[1:n-1] = j[1:n-1] \) and \( i_n < j_n \)

With this definition, any statement executed on the iteration described by \(i\) is executed before any statement on the iteration described by \(j\). This concept is included in the definition of data dependence to formally define a loop dependence as follows:

“There exists a dependence from statement \(S_1\) to statement \(S_2\) in a common nest of loops if and only if there are two iteration vector \(i\) and \(j\) for the nest, such that

1. \(i < j\) or \(i = j\) and there is a path from \(S_1\) to \(S_2\) in the body of the loop,
2. \(S_1\) accesses memory location \(M\) on iteration \(i\) and \(S_2\) accesses location \(M\) on iteration \(j\), and
3. one of these accesses is a write”

2.4.5 Distance and Direction Vectors

Another characteristic attributed to a loop dependence is the distance vector, which is expressed in terms of the number of loop iterations that the dependence crosses.

The dependence distance vector \(d(a,b)\) is defined as a vector of length \(n\) such that \(d_p(a,b) = b_p - a_p\), where \(p \in [1,n]\), where \(a\) is the iteration of a loop nest that contains the source of the dependence and \(b\) the iteration containing the sink. Note that \(a < b\) (lexicographically ordered) if and only if \(d(a,b) > 0\). Then, for example in the following code,

```
for(i = 0; i < N; i++)
  for(j = 0; j < M; j++)
    for(k = 1; k < L; k++)
      a[i+1][j][k-2] = a[i][j][k];
```

In the iteration \((0,1,3)\) the array element \(a[1][1][1]\) is written. Later, in the iteration \((1,1,1)\) the same location is read. Then, there is a true dependence from \(S(0,1,3)\) to \(S(1,1,1)\), and the dependence distance vector according to the definition is \(d(a,b) = (1,0,-2)\).

Dependence direction vectors is also a vector of length \(n\), and it is defined over the definition of distance vectors as follows:

\[
D_p(a,b) = \begin{cases} 
  "<" & \text{if } d_p(a,b) > 0 \\
  "=" & \text{if } d_p(a,b) = 0 \\
  ">" & \text{if } d_p(a,b) < 0 
\end{cases}
\]  

Again, where \(p \in [1,n]\). In the previous example, where the dependence distance vector is \(d(a,b) = (1,0,-2)\), the dependence direction vector is:

\[
D(a,b) = \{ <, =, > \}
\]

In the case that a loop index does not appear in any subscript at either the source or the sink. The distance is not constrained. Therefore, it can take any legal direction. This is represented by the symbol "=", which denotes the union of all three directions. For example in the following loop:
There is a true dependence from \(a[i+1]\) to \(a[i]\), whose direction vector is \((*,<)\). Note that there is no presence of the loop variable \(j\) in any subscript in the referenced array \(a\). When testing all possible direction the previous direction vector can be expand as \([(<,<), (=,<), (>,<)]\). The last direction vector does not mean that the dependence is illegal, but instead that there is a dependence of the opposite type in the reverse direction, then third vector actually corresponds to a level-1 anti-dependence with direction vector \((<,>)\).

The main purpose of the direction vector is to show a relation between the iteration vector at the source and the iteration vector at the sink, and with this information it is possible to determine the effect of a transformation on the code, just by looking at how the transformation changes the direction vector of the dependences contained in the loop nest where the transformation is applied.

At this point, it is important to notice that a dependence cannot exist if it has a direction vector whose leftmost non-"=" component is not "<", because this would imply that the sink of the dependence occurs before the source, which is impossible according to the definition of data dependence.

### 2.4.6 Direction Matrix

The direction matrix is defined for a nest of loops as a matrix in which each row is a direction vector for some dependence between statements contained in the nest. The following example illustrates the concept. For the code shown in Listing 2.9 the direction matrix is:

\[
DM(i,j,k) = \begin{pmatrix}
<, & <, & = \\
<, & =, & >
\end{pmatrix}
\]

Note that this definition allows identical direction vectors to be represented by the same row. Therefore, each row represents one or more dependences and due to the definition of the direction vector each column represents a level of the nested loop. In the example the first row is the true dependence from \(a[i+1][j+1][k]\) to \(a[i][j][k]\), and the second row is the true dependence to \(a[i][j+1][k+1]\).

### 2.4.7 Loop-Carried and Loop-Independent Dependences

Now that loop dependences have been fully defined, they can be classified according to the way in which the dependence arises. There are two possible ways. Let's consider \(S_2\) to depend upon \(S_1\), then:
1. $S_1$ can access the common location in one iteration, and in a later iteration $S_2$ accesses it, this is a loop-carried dependence, and it exists only when the loops are executes.

2. In the second case, in the exact same iteration $S_1$ and $S_2$, in this order, access the common location. Then, this is a loop-independent dependence, and it exits because of the position of the code within the loops.

Therefore, loop-carried dependencies determine the order in which loops must be iterated, whereas loop-independent dependences determine the order in which the code is executed within the loop nest. This classification covers any possible data dependence.

**Loop-Carried Dependence:** There is a formal relation between a loop-carried dependence and the direction vector, which can be useful to recognize them systematically.

“There is a loop-carried dependence on statement $S_1$ to statement $S_2$ if and only if $S_1$ references a location $M$ on iteration $i$, $S_2$ references $M$ on iteration $j$, and $d(i,j) > 0$ (that is $D(i,j)$ contains a “<” as its leftmost non-“=” component)”

Another important property of loop-carried dependences is the level of the dependence, which is defined as the index of the leftmost non-"=" of $D(i,j)$ for the dependence. This property is useful to determine which transformations can be applied and which need to be excluded. Due to its importance, a special notation is used to express them within the dependence. A level-$l$ dependence between $S_1$ and $S_2$ is denoted $S_1\delta_lS_2$.

**Loop-Independent Dependence:** As for loop-carried dependences, loop-independent dependences can be related with the iteration vector:

“There is a loop-independent dependence on statement $S_1$ to statement $S_2$ if and only if $S_1$ references a location $M$ on iteration $i$, $S_2$ references $M$ on iteration $j$, and $i = j$ (that is, $D(i,j)$ contains only “=” components)”

This definition states that a loop-independent dependence exists if two statements reference the same memory location in a single iteration of all the common loops. Doing an extension to the level notation utilized in loop-carried dependence, loop-independent dependence is denoted by an infinite level, that is, $S_2$ depends on $S_1$ with loop-independence is denoted $S_1\delta_\infty S_2$.

### 2.5 Data Reuse Analysis

The objective of the data reuse analysis is to identify a set of iterations that access the same data or the same cache line. Reuses can be classified according to the type of access. A static access is the access itself to an array location in the code, a dynamic access refers to the various iterations of the accesses as the loop nest is executed. Then, if the iterations reusing the same data originate from the same static access, the reuse is called self-reuse, and if they, instead, come from different static accesses, then this is a group-reuse.

Also a different classification can be done regarding the referenced location. The reuse is temporal if the same exact location is referenced; and it is spatial if the same cache line is referenced.

For clarification of these concepts, the following code is presented in [1].
The accesses $Z[j]$, $Z[j+1]$, and $Z[j+2]$ each have self-spatial reuse because consecutive iterations of the same access refer to contiguous array elements. Notice that without iterating the loop, they as a group exhibit spatial reuse, since most likely contiguous elements reside in the same cache line.

In addition, they have self-temporal reuse, since the exact same elements are used repeatedly in each iteration of the outer loop.

Although there are $4n^2$ dynamic accesses in this code, if the reuse can be exploited, it is needed to bring only $\frac{n^2}{c}$ cache lines into the cache, where $c$ is the number of words in a cache line. It is possible to save a factor of $n$ due to self-spatial reuse, a factor of $c$ due to spatial locality, and finally a factor of 4 due to group reuse.

Loops commonly exhibit data reuse, i.e., they read or write the same data elements multiple times. A data reuse is said to be realized, if a reuse results in a cache hit. By improvement of locality in the cache, which means to put closer in time the accesses to the same data or the same cache line, can increase the number of data reuse that is realized. This implies that the number of cache misses is reduced, and therefore the execution time. In summary, the improvement of locality reduces execution time by retaining data in the cache between uses in order to avoid long memory access latencies, in other words, the performance can be improved by the enhancement of locality when it increases the number of data reuse realized.

### 2.6 The CoSy Compiler Development System

The CoSy compiler development system is an environment for the creation of retargetable optimizing compilers. CoSy primary input languages are C and FORTRAN 95. CoSy's main characteristic is its modularity and flexibility that allows the construction of compilers for a wide range of architectures.

In the construction of a compiler three phases are recognized:

- **Front-end**: it consists of engines that lower the source code toward the unified IR.

- **Middle-end**: the engines transform the program from IR to IR level.

- **Back-end**: the engines in this phase use a lower representation so that the target's architecture and instruction set information is used to generate accurate machine code.

In CoSy, the intermediate representation used in the middle-end is called Common CoSy Medium-level IR (CCMIR) and the one used by the back-end engines Low-level IR (LIR). A compiler can be built using the different modules that work around the intermediate representation (IR). Every compiler functional unit is called engine, which can be shipped within the CoSy environment or handwritten.

In this thesis, there is a proof-of-concept for a selected loop reordering transformation. The proof consists of the extension of a CoSy-based compiler, thus the implementation requires the
creation of a new optimizing engine. Note, the implementation of the transformation as an engines implies that it can also be used in the construction of other compilers.

In summary, among the main properties of CoSy resides the well-defined, distributed and extensible CCMIR. CoSy includes well-tested, standard optimization engines and also gives the possibility to write extra engines which also operates on the IR. Its back-end generator is also valuable. It converts the target description language into C-code for a compiler back-end that converts CCMIR into target assembly.

2.6.1 Loop Information in CoSy

The CCMIR is the structural definition of the data structures on which the engines operate. It is a fully typed graph-based representation that can be used to represent all levels of a hierarchy of data structures, starting from the compilation unit until the primitive types like constant values and operators. The full-Structured Description Language (fSDL) is used to define the CCMIR, this specification is distributed and extensible. Each compiler engine must have a Structure Definition Language (SDL) file, describing the IR data structures that it is going to manipulate.

For the implementation of the selected loop reordering transformation engine, some specific data structures were needed, specifically the ones related with loops. This section presents the main data structures used in CoSy to handle the related loop information.

Loop Markers

Loop markers are loop annotations in the CCMIR generated by the loopanalysis engine. This structure is composed of the fields shown in Listing 2.11.

```
Listing 2.11. CoSy mirLoopMarker structure

domain
mirLoopMarker: <
  Init: mirBasicBlock,
  InitSideEffect: mirLoopSideEffectKind,
  Guard: mirBasicBlock,
  GuardSideEffect: mirLoopSideEffectKind,
  Prehdr: mirBasicBlock,
  Test: mirBasicBlock,
  TestSideEffect: mirLoopSideEffectKind,
  Body: mirBasicBlock,
  Incr: mirBasicBlock,
  IncrSideEffect: mirLoopSideEffectKind,
  Exit [primary]: XLIST(mirBasicBlock),
  Header: mirBasicBlock
>;
```

The loop marker annotation contains loop elements to relate the loop structure with the loop variables. This loop element concept is the one presented in Section 2.4.3 where CFG-loops were introduced. For defining a loop according to CoSy documentation four loop elements are mandatory: the Init, Test, Body and Incr. The Guard and Prehdr loop elements are optional.

The fields in the loop marker structure refer to the unique first basic block of the associated loop element. The Header field defines which basic block is the entry of the loop. The four
mandatory loop elements and the set of loop exit basic blocks make analysis and transformations of loops easier \cite{4}.

Loop Variables

The structure for the annotation regarding loop variables is represented in Listing \ref{lst:mirLoopVar}.

\begin{verbatim}
Listing 2.12. CoSy mirLoopVar structure

domain
  mirLoopVar: <
    mirFlds + <
      Var : mirObject,
      IsDeadAfterLoop : BOOL
    > + {
      mirLoopUpdateVar <
        UpdateExpr [primary] : mirEXPR,
      >,
      mirBasicIndVar <
        InitExpr [primary] : mirEXPR,
        UpdateExpr[primary] : mirEXPR,
        InvarExpr [primary] : mirEXPR,
      >,
      mirLoopControlVar <
        InitExpr [primary] : mirEXPR,
        UpdateExpr[primary] : mirEXPR,
        InvarExpr [primary] : mirEXPR,
        GuardExpr [primary] : mirEXPR,
        TestExpr [primary] : mirEXPR,
      >,
      mirIterCounter <
        InitExpr [primary] : mirEXPR,
        Upwards : BOOL,
        GuardExpr [primary] : mirEXPR,
        TestExpr [primary] : mirEXPR,
        IterCount [primary] : mirEXPR,
      >,
      mirHWLoopCounter <
        IterCount [primary] : mirEXPR,
      >
    >;

Note that the mirLoopVar domain has different possible instances, formally called operators, that classify the loop variables according to their kind (mirLoopUpdateVar, mirBasicIndVar, mirIterCounter, etc). The classification tries to describe the loop variables as specific as possible.

For instance, an annotated loop variable is an iteration counter of a loop (mirIterCounter in CCMIR), when the update statements are all of the form "$v = v + 1$" or "$v = v - 1$", the number of iterations are not changed by statements in the loop and the variable is tested at the
end of the loop. Due to all the constraints according to the documentation for being classified as \texttt{mirIterCounter}, the private fields to this kind of loop variable are: \texttt{InitExpr}, \texttt{TestExpr}, \texttt{GuardExpr} (in case there is a guard), the explicit number of iterations (\texttt{IterCount}). Since the increment is guaranteed to be 1, a boolean flag indicates whether it is +1 or -1.

When the update statements of the loop variable is not exactly +1 or -1, but an invariant expression, i.e. of the form \( v = v + i \) or \( v = v - i \), then the annotated loop variable is a \textit{loop control} variable. Notice that the differences in the private fields of this operator are the \texttt{InvarExpr} in \texttt{UpdateExpr}, and that the number of iterations is not annotated.

The \texttt{mirBasicIndVar} operator identifies the \textit{basic induction} variables of the loop. This are variables that are incremented or decremented by some amount in each iteration but they are not used to test the end of the loop. Therefore, the private fields does not contains a \texttt{TestExpr}.

The \texttt{mirLoopUpdateVar} is the most general loop variable identified, there are no further restrictions, the variable should not be initialized in the \texttt{Init}, should not be used to test the end of the loop, only an update expression is identified and it is annotated in the \texttt{UpdateExpr} field. And finally a \textit{hardware loop counter} (\texttt{mirHWLoopCounter}) represents a loop variable that is the iteration counter of a CCMIR hardware loop construct. The last two operators are cases where the loop reordering transformations can not be applied.

\textbf{Loop-carried Dependence Edge}

The last CCMIR element, \texttt{mirLcdEdge}, is shown in Listing 2.13. It represents the edges of the data dependence graph that a CoSy-standard engine, \texttt{lctepana} attaches to the \texttt{mirLoopMarkers}.

\begin{verbatim}
Listing 2.13. CoSy \texttt{mirLcdEdge} structure

domain
\texttt{mirLcdEdge} [ref] : <
  \texttt{depKind} : \texttt{DEPKIND},
  \texttt{ObjectLabel} : \texttt{NAME},
  \texttt{Source} : \texttt{mirSTMT},
  \texttt{Sink} : \texttt{mirSTMT},
  \texttt{AnyLevel} : \texttt{BOOL},
  \texttt{Independent} : \texttt{BOOL},
  \texttt{Level} : \texttt{INT}
> + {
  \texttt{lcdFromTop} <>,
  \texttt{lcdScalar} <>,
  \texttt{lcdSubscript} <,
    \texttt{DirVec} : \texttt{SEQ(DEPDIR)}
  }
};

\end{verbatim}

In a DDG, the edges represent the dependences, which can be either a true, anti- or output dependence. This information is annotated in the \texttt{depKind} share field. The statements that constitutes the source and the sink are also stored. In addition, the level of loop nesting at which the dependence is carried is also captured:

- \texttt{Any Level}: the dependence is carried in all loop nest levels
- \texttt{Independent}: the dependence is not carried by any loop
- \texttt{Level}: the dependence is carried by the loop nest specified in this field
2.6 The CoSy Compiler Development System

Figure 2.5. Example of loop markers tree structure

The edges of the dependence graph can either be for simple scalar objects or for subscripted array accesses. This differentiation is made by means of the operators `lcdScalar` and `lcdSubscript`. The remaining is an auxiliary edge from the dummy top node of the data dependence graph.

A dependence direction vectors is stored for subscripted array accesses. Giving the direction vector for each loop nest that can be any combination of:

- DEPDIR_LEFT: the source is in an earlier iteration than the sink.
- DEPDIR_EQUAL: the source is in the same iteration than the sink.
- DEPDIR_RIGHT: the source is in a later iteration than the sink

2.6.2 CoSy Engines

From the set of engines shipped with CoSy, here are mentioned the ones of special interest for the understanding in the following chapters:

- **mirtoc**: stands for Medium-Level IR to C engine, it converts from CCMIR to C code. The c-to-c compiler is said to be mirtoc based, since this engine is the one that brings code back to the original source language after any transformation at the CCMIR level.

- **loopanalysis**: it annotates loop with information such as the basic blocks involved and its role, the number of iterations, the increment step, list of loop variables and so on. The engines analyzes every procedure in a program. It identifies the loops and it organize them in a tree with the root attached to the procedure itself. In Figure 2.5 it is illustrated how a loop marker tree looks like.

- **lcdepana**: it builds a graph with dependency edges between statement pairs. For each loop nest structure, the dependence graph is attached to the outermost LoopMarker of the analyzable loop nest. In Figure 2.5, an example of a possible distribution of the information of the data dependences for each loop nest is depicted with red rings around the
loop markers. Then, the loop markers has attached the data dependence graph structure. The ldcepana engine extends the loop marker definition presented in Section 2.6.1 with a boolean variable ldoValid in order to notify if a loop marker has a DDG attached or not. Some loops cannot be analyzed by the engine. This is caused by control-flow and/or statements with data accesses that prevent the calculation of conflicting memory references. Thus, there are a number of restrictions on the input code.

A statement inside a loop nest structure cannot be:

- a function call (mirCall in CCMIR);
- a function call with return values (mirFuncCall);
- a function call with multiple return values (mirMultiResCall);
- an if-statement (mirIf);
- a switch-statement (mirSwitch).

Additionally, the loop can only be a while-do loops, the init, test, and update expression of the loop cannot have any side-effects, such as for example post-increment or access to volatile variables.

The integration of an engine within a compiler is done by means of the Engine Description Language (EDL) file. In this work, it is performed the development and integration of an engine within a compiler. Therefore, a modification of EDL files is needed. A EDL file describes not only a hierarchy, but the interactions and calling parameters among the engines that constitute a compiler. For instance, the engines can be specified to run sequentially, in parallel, in a pipeline or in a loop, also the evaluation of conditions is part of the possible dynamics.

2.7 Related Work

Since loop transformation are part of the code optimization field, the research in loop transformations follow the same line, where models and algorithms are proposed in order to determine one or more application properties to solve the problem of how to apply these transformations. Special attention has been put on loop permutation and tiling ([6, 7, 12, 15, 18, 21]), that's why the remainder of this work is focusing on these two transformations. Other kinds of techniques such reversal and skewing have been viewed as transformations that enable loop permutation [7]. Therefore, their behavior is also analyzed in next chapter. A representative set of studies help to exemplify the different approaches to solve specific problems of the application of loop transformations.

Most of the studies are focused only in the formulation of algorithms to select and configure the loop transformations so that improvement of the cache behavior is achieved. For instance, Wolf and Lam in [21] proposed an algorithm that improves the locality of a loop nest using a mathematical formulation of reuse and locality and a loop transformation theory that unifies various transformations as uni-modular matrix transformations.

Sarkar [19] describes a transformer that performs automatic selection of loop transformations using a cost-based framework. It estimates the memory cost in terms of the number of cache lines and the number of memory pages accessed when executing a loop. It considers the different memory layouts, the size of the memory and the occurrence of capacity and conflict misses.
McKinley et al. [7] propose a compound algorithm to select loop organizations according to a simple cache model, which handles conflict misses in a simpler manner. The algorithm assigns a cost in terms of the number of accesses to memory using a data reuse analysis and the cache model. With this Loop Cost algorithm an analysis of the possible improvement of locality can be done. Since the main goal of loop transformations is the enhancement of locality, the algorithm is used in the next chapter to analyze the locality improvements obtained. Hence, the algorithm is shortly described in the next section.

The work done by Zhao et al. in [23] has a close relation with the main objectives of this thesis. It treats specifically the profitability property. Their study describes a framework for predicting the impact of loop transformations on the cache performance. It is a predictive-analytic approach that uses three models to describe code, transformations and resources.

- The code model expresses the code characteristics that affect the cache, which are the loop's headers and the sequence of array references in a loop body.

- The transformation model describes the impact of a specific transformation on the original code. The study presents the impact function for the main loop transformations: loop interchange (permutation), loop unrolling, tiling, reversal, fusion, and distribution.

- The resource model consists of a model of the cache behavior that estimates the cost of executing a code segment in terms of access to the memory. This model is an improvement of the model presented in [8]. It includes a simplified version of the algorithm presented in [9] so that all possible sources of conflict misses are reflected.

They showed that predictions can be used to selectively apply a loop transformation based on the cache and the loop configurations. It can also be used to select the transformation with greatest profit among several applicable ones for a particular code context. Conclusions and comments about future work show the complexity in integrating all factors influencing optimizations to a single model. Their job is also more suitable for x86 processors. The resources models depend on the machine architectures and may also need to be combined to make more accurate predictions, and the interaction among the transformations also needs to be taken into account [24].

From these few examples, it is seen that loop transformations have been theoretical treated, focusing in the creation of new models for the calculation of the profitability, order and configuration of one or a set of transformations. However, most of the proposed models use only cache cost models and are focused on techniques for improving data locality, but it is not clear how to predict their impact on the performance of the whole system. The integration in a single framework of all needed models for describing all possible factors influencing is hard to achieve, which also motivates the use of a machine learning as a technique to create these complex models.

### 2.7.1 Loop Cost Algorithm

S. Carr et al. in [7] propose a data reuse analysis to choose what they called the “best order” of loop index in terms of locality for a particular loop nest. For doing that, they “quantify the locality” by counting the number of cache lines that need to be accessed when a particular loop index is placed as the innermost loop. The main principles and results of this algorithm will be applied in a particular sample code in the evaluation done in Chapter 4, so that a “quantification of the locality” can be obtained.
With respect only to the innermost loop index, if a variable is invariant, then it exposes temporal reuse, and it needs to be loaded only once. Instead, if a static access exposes spatial reuse, it is because, it calls dynamically consecutive variables in memory. Since they are close to each other, then it is not necessary to load a new variable at every iteration of the innermost loop, but instead \( \text{trip} \times \frac{\text{element size}}{\text{cache line size}} \) times, where \( \text{trip} \) is the number of iterations of the innermost loop. And finally when no reuse is exposed by a variable, then the cache would need to be accessed on every iteration of the innermost loop, i.e. \( \text{trip} \) times.

Any set of references that expose group-reuse are called reference group (\( \text{RefGroup} \)). The basic principle of the algorithm \( \text{Loop Cost} \) is to calculate the cost in terms of cache lines of each reference group \( \text{Ref}_k \) with \( k \in [1, m] \), where \( m \) is the number of reference groups in the code. The cost is calculated according to the data reuse exposed with respect to each loop index when they are placed as innermost loop (\( l_i \)), see Equation 2.5.

\[
\text{LoopCost}(l_i) = \sum_{k=1}^{m} (\text{RefCost(Ref}_k, l_i)) \times \prod_{h \neq l_i} \text{trip}_h 
\]  

Equation 2.5

The cost values are the ones mentioned above and summarized in Table 2.2.

<table>
<thead>
<tr>
<th>Data Reuse</th>
<th>RefCost(Ref(_k), (l_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal</td>
<td>(1)</td>
</tr>
<tr>
<td>Spatial</td>
<td>(\text{trip} \times \frac{\text{cls}}{\text{stride}})</td>
</tr>
<tr>
<td>No Reuse</td>
<td>(\text{trip})</td>
</tr>
</tbody>
</table>

Table 2.2. Locality cost in terms of cache lines

A variable exposes temporal reuse when it is invariant with respect to \( l_i \). In the case of spatial reuse, it is important to consider that C language uses a convention of row-major order for the data arrays. Therefore, whenever the rightmost subscript depends only on \( l_i \), and it is the only one that depends on \( l_i \), the variable is exposing spatial reuse. And when there is no spatial nor temporal reuse then, “no reuse” is inferred. Once the classification is done, the Table 2.2 can be used to obtain the values of \( \text{RefCost} \), and finally the \( \text{LoopCost} \) simply adds the cost of all \( \text{RefGroups} \) with respect to the loop analyzed (innermost loop) and then multiplies by the \( \text{trip} \) values of all the remaining loops (outer loops).

\[\text{cls} \] in Table 2.2 is the cache line size in terms of the type of elements access, and also notice that the stride also reduced the benefits of the spatial reuse (for example, when \( \text{step}_i \neq 1 \)).
Chapter 3

Evaluation of Loop Reordering Transformations as C-to-C Transformations

In this chapter, the results of applying loop reordering transformations at the source level are presented. Loop permutation and tiling were both applied manually to well-known examples written in C, which are later compiled for three different uni-processors. The performance of the execution of the original and the transformed version of the code for each architecture is measured and the results are analyzed. Additionally, enabling transformations like reversal and skewing were also evaluated. The purpose is to determine if these transformations, applied at source level, generate improvements on the global performance of an application, and to determine the configurations that might change the behavior of the performance. As a second objective, the influence of some factors inherent to the program and to each architecture that might be impacting the profitability of the transformations are analyzed.

Loop permutation and tiling are loop reordering transformation for mainly the improvement of the locality. Compilers either exclude these optimizations entirely or the application of these transformations is based on a fixed criteria, usually a conservative heuristic that in most of the cases is tuned by looking at the cache memory characteristics (size, miss rate, cache line) and other architectural features.

The performance of an application is affected by the cache behavior, but it is also affected by other factors like the instruction scheduling, register allocation or loop overhead. Therefore, the improvements of the performance when applying one of the considered transformations is not guaranteed. In this chapter, it is studied under which circumstances the application of a particular transformation is more likely to be beneficial and which patterns or factors of a particular program result in variations of the global performance.

The evaluation of each transformation is organized as follows. First, the definition and a simple example of the transformation is given. Then, the associated legality test is briefly explained, and the theoretical basis for the analysis of the profitability is summarized. Using this basis, the results of the experiments are presented and analyzed.
3.1 Simulation Environment: Memory Characteristics and Tools

The architectures considered for the analysis are ARM926EJS, Texas Instruments C6455 and Tensilica Diamond 570T. For measuring the performance, cycle accurate simulators of the three processors were used. In case of the ARM, it was used the simulation tool Synopsys Platform Architect F-2011.06-SP1 and specifically the core model ARM926EJS_AHB_CARBON. In the case of TI-C6455, Code Composer Studio version 4.2.4 was the development environment and the MegaModule C6x+ the simulation core. And finally, for Tensilica Diamond 570T, the xt-sim tool of the Xtensa Tool Version 8.0 was used. For each of them, the memory subsystem was configured as shown in Table 3.1.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>ARM926EJS</th>
<th>Diamond 570T</th>
<th>TI-C6455</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Cache</td>
<td>32 KB</td>
<td>16 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>I-Cache</td>
<td>32 KB</td>
<td>16 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
<td>2-way</td>
<td>2-way</td>
</tr>
<tr>
<td>Cache Line</td>
<td>8 words</td>
<td>16 words</td>
<td>16 words</td>
</tr>
<tr>
<td>Miss Latency</td>
<td>3 cycles</td>
<td>3 cycles</td>
<td>2 cycles</td>
</tr>
</tbody>
</table>

3.2 Loop Permutation

3.2.1 Definition

Also called Loop Interchange, this transformation exchanges the position of two loops in a perfect loop nest, with no other changes. For example:

**Listing 3.1. Example 1 - Original**

```plaintext
for(j = 2; j < N; j++)
    for(i = 2; i < M; i++)
        a[i][j] = a[i][j] * 2;
```

**Listing 3.2. Example 1 - Permuted**

```plaintext
for(i = 2; i < M; i++)
    for(j = 2; j < N; j++)
        a[i][j] = a[i][j] * 2;
```

The previous definition loop permutation considers only the interchange of two loops. However, any desired loop order in a loop nest can be achieved by interchanging pairs of loops several times. This is also understood as loop permutation. It achieves performance enhancement by improving cache performance and exposing parallelism. Loop permutation might be used as follows:

- **Move an independent loop to the innermost level**: vectorization, which is a fine-grained parallelism, is enabled by interchanging an inner dependent with an outer independent loop.

- **Move an independent loop to the outermost level**: the purpose is to pull in the loops that actually carry the dependences, in order to increase the granularity of each iteration. Hence, a coarse-grained parallelism is exposed.
3.2 Loop Permutation

- **Transform to stride-1 accesses**: loops are swapped so that the inner loop traverses the array according to the actual memory layout, in order to reduce cache misses.

- **Increase the number of loop-invariant expressions in the inner loop**: this reduces the number of accesses to the memory.

The configuration of a transformation refers to the parameter that can be used to tune the application for obtaining the best profitability, considering the factors previously exposed. To fully specify the transformation desired, the only parameter needed is the order of the loops.

### 3.2.2 Legality Test

Since loop permutation is a reordering transformation, it has a legality test associated. A reordering transformation fully preserves the meaning of a program if it fully preserves every dependence in the program, and this can be checked by using the dependence direction vectors.

In general, it was stated that a transformation is **legal**, if after it has been applied, none of the dependence direction vectors of the loop nest have as first non-"=" component a ">", i.e., the direction vectors must be lexicographically positive. Therefore, by knowing how the transformation modifies the direction vectors it is possible to tell if it is legal without actually applying it.

In the case of loop permutation, being \( D(i) \) the direction vector for a dependence in a loop nest of \( n \) loops, the direction vector for the same dependence after the transformation is determined by applying the same permutation to the elements of \( D(i) \). For example, let

\[
D(i) = (d_1, d_2, ..., d_p, ..., d_q, ..., d_n)
\]

and assume that the loops \( p \) and \( q \) in the perfect nest of \( n \) loops are interchanged, the dependence vector \( D(i) \) becomes \( D'(i) \), where

\[
D'(i) = (d_1, d_2, ..., d_q, ..., d_p, ..., d_n)
\]

The definition of legality can be extended to the dependence direction matrix. By permuting columns in the direction matrix one can reflect the effect of permuting the corresponding loops on all the dependences, instead of just one. If after permuting the columns there is no ">" direction as the leftmost non-"=" direction in any row, then the permutation is legal.

### 3.2.3 Profitability of Loop Permutation

The different uses of loop permutation generate different benefits that can cancel out each other, hence the resultant profitability is uncertain. The objective in this section is to list the different uses and the circumstances in which it is likely to obtain a profit.

As it was pointed out previously, loop permutation can achieve improvements in performance, by either the enhancement of the memory hierarchy performance or by exposing parallelism.
Exposing Parallelism

Loop permutation exposes parallelism by reallocating the dependences. The resultant dependence pattern can be inferred by analyzing the modifications in the direction matrix.

The main factor that determines if a particular interchange pattern is profitable is the architecture of the target machine \cite{14}. When applying a specific permutation, there is a movement of dependent loops in or out, and depending on the architecture in which it is accomplished, it might expose some parallelism. How much benefit from the realization of the exposed parallelism is added over the original loop order, depends on factors like the number of functional units available or the degree of granularity obtained. The following example shows how loop permutation can expose parallelism depending on the architecture.

Listing 3.3. Example 2 - Original
\begin{verbatim}
for(i = 0; i < N; i++)
  for(j = 0; j < M; j++)
    for(k = 0; k < L; k++)
      a[i][j+1][k+1] = a[i][j][k] + 2;
\end{verbatim}

Listing 3.4. Example 2 - Permuted
\begin{verbatim}
for(k = 0; k < L; k++)
  for(j = 0; j < M; j++)
    for(i = 0; i < N; i++)
      a[i][j+1][k+1] = a[i][j][k] + 2;
\end{verbatim}

In the original code shown in Listing 3.3, there is a true dependence with a dependence direction matrix \( DM(i, j, k) = (=, <, <) \). The dependence is carried by the \( j \)-loop. Applying permutation, as shown in Listing 3.4 modifies the dependence direction matrix to \( DM(k, j, i) = (<, <, =) \). Note that with this change, the dependence is now carried by the outer loop \( k \), which might be beneficial since vectorization is enabled. The two inner loops could be vectorized. This kind of parallelism is actually useful on vector machines, and machines that leverage the instruction-level parallelism, such as VLIW and superscalar processors.

On the other hand, if the code is executed on a multiprocessor machine, then the original code is more likely to perform better, because the loop \( i \) can be executed in parallel since it is an independent loop. In general, for parallel machines the profit generated for exposing certain parallelism depends on factors like the degree of granularity and the overhead of the synchronization. There should be a trade-off between the load balance and the communication and synchronization cost.

Memory Hierarchy Improvement

Loop permutation can impact in a greater degree the performance of the register level and the cache memories within the usual organization of the memory hierarchy.

Register Reuse

If a referenced array element is invariant within the innermost loops, it can be loaded into a register before the inner loop, then it is said that the register is reuse during the execution of the loop. The benefit behind register reuse lays in the fact that it reduces the number of load/store operations. The access to the memory might take long even in case of a cache hit. Therefore, the enhancement in performance that can be achieved by the improvement of the register reuse might be important.

There are two techniques in which this can be achieved. The first one is by means of analyzing the dependence pattern using the direction matrix, and the second one is through a stride analysis.
The idea behind the maximization of the register reuse by means of the modification of the dependence pattern, is to get the loop that carries the most dependences to the innermost position. This makes it possible to reuse values by keeping them in registers [14]. Since this mechanism for improving the register reuse is also attached to the dependence pattern, depending on the architecture, this can create conflicts with the mechanism for exposing the parallelism. The following example illustrates how this modification of the dependences can enable the register reuse.

Listing 3.5. Example 3 - Original
for (i = 1; i < N; i++)
    for (j = 0; j < M; j++)
        r = a[i-1][j];
        a[i][j] = r;

Listing 3.6. Example 3 - Transformed
for (j = 0; j < M; j++)
    r = a[0][j];
for (i = 1; i < N; i++)
    a[i][j] = r;

In the original code, the number of loads and stores is \((N - 1)M\). Note that the value of the first row is propagated to the second and the third and so on, therefore the value on each column is the same, but anyways all load and stores are performed. By interchanging the loops the matrix is traversed by one column at a time. This make it possible to load only the first value and keep it in a register during the iteration of the inner loop. This transformed code still requires \((N - 1)M\) stores but the number of loads is reduced to \(M\).

The way that the dependence direction matrix can be used for this purpose, is explained in [14] as follows. First, one needs to identify the rows corresponding to true dependences that have only one “<”, with the remainder of the positions containing “=”. The loops containing these dependences are the ones considered to be moved to the innermost position. Consider a code that has the following direction matrix:

\[
\begin{pmatrix}
< & = & = \\
< & < & = \\
= & < & < \\
\end{pmatrix}
\]

The first and the third row, are the ones that might lead to some register reuse. The dependence described by the second row would never be carried by the innermost loop, therefore register reuse cannot be enabled for this dependence. The outer or the second outer loop are candidates to be pulled in. If the outer loop is moved towards the innermost position, the dependence corresponding to the first row might lead to a register reuse.

The second technique mentioned, the one related with the stride analysis works as follows. At a register level, the register reuse can also be seen as the conversion of temporal data reuse into temporal locality. With loop permutation the stride of the memory access changes. If the number of loop variables used to reference a memory access is less than the depth of the nested loop, then there is a loop order that exposes a stride-0 access. This correspond to a register reuse. The situation can be identified in a direction matrix by the presence of an “*” element.

On machines with cache memories, an interchange that improves register reuse by creating stride-0 access, may also change a stride-1 access pattern to a stride-n access pattern with much lower overall performance due to the increase of cache misses.

Cache Performance

The improvement of the cache performance is achieved by the conversion of the data reuse into data locality. Loop permutation is a transformation that enables this conversion, and enhances
locality by reducing the number of iterations between uses of the same data.

One way to measure the profitability of loop permutation for the cache performance is by means of the reduction of the number of cache misses, comparing the ones occurred in the original code with the ones in the transformed code. What matters for a transformation to be profitable is the difference between the cache misses occurred in each version of the code.

For any given order of loop indexes in a nested loop, the number of compulsory misses remains the same. They correspond to the number of different cache lines that are accessed by all the array references when the complete loop nest is executed. Therefore, when no conflict nor capacity misses are generated, the transformation does not generate any benefit.

The generation of capacity or conflict misses depends on the problem size, and on all the characteristics that fully define a cache memory such as: size, associativity, cache line size, replacement policy, etc. For example, if the cache is fully-associative then no conflict misses can occur, therefore only the presence of capacity misses determines the profitability of a permutation. A certain loop order might improve the performance if the iterations that access the same data or the same cache line are performed closer to each other, so that when they are evicted from cache they are no longer needed.

With the following example two ideas are illustrated:

1. If no conflict nor capacity miss is present, the profitability of any transformation is zero.

2. The presence of these types of misses depends on the cache characteristics, and the problem size.

Now let’s consider the following transformation and a fully-associative memory with cache line size $L_s$, and size $C_s$.

Listing 3.7. Example 4 - Original

```c
for ( i = 0; i < M; i++)
    for ( j = 0; j < N; j++)
        c[j][i] += a[i] * b[j];
```

Listing 3.8. Example 4 - Permuted

```c
for ( j = 0; j < N; j++)
    for ( i = 0; i < M; i++)
        c[j][i] += a[i] * b[j];
```

Considering the code of Listings 3.7 and 3.8 it is shown in Table 3.2, the number of cache lines accessed by all references only during the first iteration of the outer loop if $l_{inner}$ is placed as innermost loop.

Table 3.2. Loop Cost to derive the “best” loop order

<table>
<thead>
<tr>
<th>$l_{inner}$</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref&lt;sub&gt;k&lt;/sub&gt;</td>
<td>(c[j][i])</td>
<td>(\frac{\text{trip}_j}{L_s})</td>
</tr>
<tr>
<td>a[i]</td>
<td>(\frac{\text{trip}_i}{L_s})</td>
<td>(1)</td>
</tr>
<tr>
<td>b[j]</td>
<td>(1)</td>
<td>(\frac{\text{trip}_j}{L_s})</td>
</tr>
<tr>
<td>Total</td>
<td>(\frac{2\text{trip}_i}{L_s} + 1)</td>
<td>(\text{trip}_j + \frac{\text{trip}_j}{L_s} + 1)</td>
</tr>
</tbody>
</table>

Since a fully associative cache is assumed, for the original code ($l_{inner} = j$) cache misses exist depending on the size of the problem, that means the number of iterations of $j$ ($\text{trip}_j$).
If the number of different cache lines, if $j$ is fully executed as innermost loop, is greater than the total number of cache lines available in the memory, i.e. \( \frac{(L_s + 1) \cdot \text{trip}_j}{L_s} + 1 > \frac{C}{L_s} \), then some of the cache lines retrieved for the first iteration of the outer loop are evicted. If the references exposed some data reuse in the outer loop, this reuse is not realized and the total number of cache lines retrieved during the execution of the complete loop nest in the original code is then given by the number of cache lines calculated for the innermost loop multiplied by the trip count of the outer loops, i.e.:

\[
CM_{\text{original}} = \left( \frac{(L_s + 1) \cdot \text{trip}_j}{L_s} + 1 \right) \cdot \text{trip}_i
\]

Note that in a cache where the replacement policy is Least Recently Used (LRU), the number of cache lines retrieved actually corresponds to the number of cache misses (CM). If only some of the cache lines are evicted in the first iteration, the execution of the second iteration evicts the rest. When the trip$_j$ is big enough the realization of the data reuse of the outer loops is prevented and the multiplication by the trip count of the outer loops is a valid approximation of the cache misses. For the transformed code a similar analysis can be done, which gives that if \( \frac{2 \cdot \text{trip}_j}{L_s} + 1 > \frac{C}{L_s} \), capacity misses are present and the number of cache lines retrieved is given by:

\[
CM_{\text{transformed}} = \left( \frac{2 \cdot \text{trip}_j}{L_s} + 1 \right) \cdot \text{trip}_j
\]

Comparing original and transformed code, which one is greater depends on the relation between the trip counts (\( \text{trip}_j = N \) and \( \text{trip}_i = M \)) of each loop. Let’s consider \( M = N \), then both expression can be expressed as follows:

\[
CM_{\text{original}} = \left( \frac{(L_s + 1) \cdot N}{L_s} + 1 \right) \cdot N
\]

\[
CM_{\text{transformed}} = \left( \frac{2N}{L_s} + 1 \right) \cdot N
\]

This shows that the number of cache misses is different if capacity misses are present, and the presence depends on the size of the problem. If there is a reduction of the cache misses from the original to the transformed code, then it is concluded that the transformation is profitable.

Now, in case that capacity misses are not present, i.e. the cache lines retrieved in the first iteration of the innermost loop fit in the cache, then the data reuse exposed by a reference in the outer loop can also be realized. Thus, the number of cache lines retrieved in case of the original code is given by:

\[
CM_{\text{original}} = (\text{trip}_i \cdot N) + (\text{trip}_j \cdot 1) + (1 \cdot \frac{\text{trip}_i}{L_s})
\]

Note that, this expression does not multiply the data reuse of each group reference by the trip count of the outer loops, as done in the LoopCost algorithm, but it multiplies it by the data reuse exposed by the same reference in the outer loop. This corresponds to the compulsory misses. Each term in the previous expression correspond to the number of cache lines retrieved for each reference, \( c[i][j], a[i], b[j] \) respectively. The terms show that each element of the arrays is accessed once even when they are used more than one time during the execution of the loop nest. This implies that the data reuse has been fully realized. The same is done for the transformed code, obtaining:
3.2 Loop Permutation

\[
CM_{\text{transformed}} = (\text{trip}_i \ast \frac{\text{trip}_i}{L_s}) + (1 \ast \frac{\text{trip}_i}{L_s}) + (\frac{\text{trip}_i}{L_s} \ast 1)
\]

Since the expression is exactly the same the reduction in cache misses when applying the transformation is zero. Therefore, when considering only capacity misses, one can say that it exists a break-even-point where the best loop order gives benefits over any other possible order. This break-even-point depends on program context factors such as the number of references, the order in which they are accessed, the trip count values, the depth of the nest, and also the memory characteristics like cache size and cache line size.

When changing the assumed cache to a direct-mapped or k-way associative memory, then there is the possibility of having conflict misses. The presence of capacity misses can also be predicted in this kind of caches by the calculation of the break-even-point where the problem size is bigger than the capacity of the cache. However, the presence of specifically conflict misses is independent from the cache capacity. It is influenced by program context factors like the cache inference among the arrays, which depends on the relative positioning of different data structures. Also when traversing an array with n-stride, if the cache line step is a multiple of the number of sets, this can lead also to a situation of high occurrence rate of cache misses. The associativity is a characteristic that alleviates this kind of situation. In general, the greater the associativity, the less the likelihood of conflicts.

From the model proposed by Ghosh et al. [9], it is stated that conflict misses are highly sensitive to slight variations in problem size and base addresses. Here, some of the factors that influence the occurrence of conflict misses are extracted, in order to identify in the future evaluation patterns where the cache misses vary.

Each array reference accessed within a loop, needs to be mapped to a set in the cache. This mapping, according to the study [9], is given by the following equation:

\[
\text{CacheSet}_{R_A}(i) = \lfloor \frac{\text{Mem}_{R_A}(i)}{L_s} \rfloor \mod N_s
\]  

(3.1)

where \(\text{Mem}_{R_A}(i)\) is the memory address accessed by the reference \(R_A\) at iteration \(i\), and \(N_s\) is the number of sets.

The relative position of the references accessed within one loop iteration, depends on the stride and the array size of the reference. In general, due to the modulo operation involved in the mapping, if there are common factors between the stride in terms of cache lines and the number of sets, then the total number of different cache sets used when traversing the loop is reduced and cache misses are more likely to be generated.

The greatest common divisor (gcd) between the number of sets \((N_s)\) and the array size in terms of cache lines \((N_{L_s})\) represents a factor of reduction in the number of sets used, from the total available. Note that since \(gcd(N_s, \frac{N}{L_s}) < \min(N_s, \frac{N}{L_s})\), the set of possible reduction factors \(RF\) can be inferred, from a specific value of cache sets.

According to the study [9], the precise definition of the relationships among the loop indexes, array sizes and base addresses, and the cache parameters to fully describe the conflict cache misses in a loop nest is complex. However, in general, from the program the following factors are influencing: relative positioning of the data structures that are referenced within the loop, the relation between the strides in which every reference is traversed and the number of sets and the associativity.

The following three points summarize the ideas regarding the improvement of cache performance achieved by loop permutation:
• Although the LoopCost algorithm only gives the upper bound for the cache misses that might be obtained, it is useful to determine if the new loop order might result in an improvement of locality. If not, even if the cache misses are present no profit is expected.

• The presence of conflict misses depends on the layout of the cache memory. If only capacity misses are considered, there is a break-even-point when increasing the problem size where improvements are expected. This break-even-point depends on program context factors like the trip counts, the number and order of references accessed and the depth of the nest. Also it depends on cache characteristics like the cache line size and the cache size.

• The occurrence of conflict misses is independent of the capacity of the cache, and it can lead to the non-realization of data reuse. The presence of conflict misses has not been fully modeled. The model proposed in \[9\] shows the complexity of the problem. Some of the factors that influence the occurrence of conflict misses have been identified, for instance, the array sizes that determine the stride in which a reference might be traversed. The relation of this stride with the number of sets and the associativity determines if the data reuse can be realized. Also the relative positioning of the data structures might lead to cache inferences and therefore a high occurrence rate of conflict misses.

3.2.4 Source Code for the Evaluation of Loop Permutation

For this transformation two sample codes were used: 3-way nesting and 2-way nesting. The first one, is the most frequently used example in literature when introducing a new algorithm that applies permutation as an optimization within a compiler (see for example \[7; 13; 22\]). The code is the computational kernel of a matrix multiplication. It has three nested loops operating over three different 2-dimensional arrays.

<table>
<thead>
<tr>
<th>Listing 3.9. 3-way nesting-Original</th>
<th>Listing 3.10. 3-way nesting-Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(i = 0; i &lt; N; i++)</code></td>
<td><code>for(i = 0; i &lt; N; i++)</code></td>
</tr>
<tr>
<td><code>for(j = 0; j &lt; N; j++)</code></td>
<td><code>for(j = 0; j &lt; N; j++)</code></td>
</tr>
<tr>
<td><code>for(k = 0; k &lt; N; k++)</code></td>
<td><code>for(k = 0; k &lt; N; k++)</code></td>
</tr>
<tr>
<td><code>c[i][j] += a[i][k] * b[k][j];</code></td>
<td><code>c[i][j] += a[i][k] * b[k][j];</code></td>
</tr>
</tbody>
</table>

For the analysis of the profitability of the transformation the dependence pattern is needed. Thus, Figure [3.1] presents the loop-carried dependences of the matrix multiplication code. The statement within the nested loop, has true, anti- and output dependences on itself, carried by the k-loop. They share the same representation as a direction vector. Therefore, the dependence direction matrix consists of a single row.

\[D(i,j,k) : \begin{pmatrix} = & = & < \end{pmatrix}\]

The second one, 2-way nesting, is a simpler example: a 2-nested loop operating over a 2-dimensional array and a 1-dimensional array. This particular example is used in \[8\], to show that the benefits obtained from the register reuse and cache performance improvement, conflict each other and the overall improvement of the application can not be guaranteed.
3.2.5 Results

In Figure 3.2, the speedup when applying loop permutation on 3-way nesting and 2-way nesting is presented. The trip count $N$, which also corresponds to the dimensions of the arrays referenced within the loop, has been varied from 0 to 1000 for 3-way nesting and until 2000 for 2-way nesting in steps of 100.

ARM926EJS

As it was mentioned at the beginning of the Section 3.2.3, one of the factors that determines if a particular interchange pattern is profitable or not is the architecture of the target machine. The ARM926EJS has a scalar architecture that does not leverage ILP or parallelism. Therefore, here loop permutation is focused on the improvement of the memory hierarchy. The results in this case are drawn with a solid line in Figure 3.2. When increasing the value of the trip count, there is a point in the graphic where the speedup becomes beneficial. For 2-way nesting this value is $N=1050$, and for 3-way nesting it is $N=650$.

First, the effect of the transformation on the register level is analyzed, to determine if the transformation can enable register reuse. For doing so, the modification done on the dependence pattern of each sample code is evaluated. For 3-way nesting, the loop order changes from $ijk$ to $ikj$, therefore the dependence direction matrix changes:

\[
\begin{array}{c}
\delta_k^{-1} \\
\delta_k \\
\delta_k^0
\end{array}
\]

\[
c[i][j] = c[i][j] + a[i][k] \ast b[k][j]
\]

This means that by applying the transformation some register reuse might be lost, giving no profit for the transformation, when considering only this possible benefit. The same analysis can be done for 2-way nesting. The modification of the direction matrix is as follows:

In the case of the original sample code, presented in Listing 3.11, the dependence direction matrix also corresponds to a single row:

\[
DM(i,j) = ( < * )
\]
Figure 3.2. Speedup when permuting on (a) 3-way nesting and (b) 2-way nesting.
3.2 Loop Permutation

\[( < * ) \rightarrow ( * < )\]

In this case, it is not possible to state anything about the register reuse by just looking at the modification of the pattern, since the dependence is carried either by the outer or the inner loop. However, an analysis of the modification of the stride is also performed. When permuting the loop nest, the stride of each variable accessed is modified as shown in Table 3.3.

Table 3.3. Stride analysis when applying loop permutation on 2-way nesting code

<table>
<thead>
<tr>
<th>Variable</th>
<th>Original Stride ((l_{\text{inner}} = j))</th>
<th>Transformed Stride ((l_{\text{inner}} = i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>total([i])</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(a[j][i])</td>
<td>(N)</td>
<td>1</td>
</tr>
</tbody>
</table>

With this analysis one can see that the original stride-0 on total\([i]\) is lost. Therefore, the register reuse in 2-way nesting, has also been lost, and then the execution of loads and stores are increased. This was corroborated by looking at the assembly code. The compiler generated the code presented in Figure 3.3.

In the original code, the variable total\([i]\) is saved in a register (r12), see address 7b8, which is reused during the \(N\) iterations of the innermost loop, which are the lines that goes from address from 7c4 to 7d4. When applying the transformation, and therefore increasing the stride to 1, the placement of the element in a register is no longer possible. Instead, the variable needs to be loaded and stored inside the innermost loop, lines 7c0 and 7cc. This implies that the code inside the innermost loop increases, therefore the execution time increases. How much the performance is degraded depends on the cost of the load and store operations, and the size of the code within the loop. Table 3.3 shows the stride analysis for 3-way nesting.

Table 3.4. Stride analysis when applying loop permutation on 3-way nesting code

<table>
<thead>
<tr>
<th>Variable</th>
<th>Original Stride ((l_{\text{inner}} = k))</th>
<th>Transformed Stride ((l_{\text{inner}} = j))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a[i][k])</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(b[k][j])</td>
<td>(N)</td>
<td>1</td>
</tr>
<tr>
<td>(c[i][j])</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

There is a loss of locality on \(c[i][j]\), but also \(a[i][k]\) wins temporal locality with the transformation. Note that \(c[i][j]\), as total\([i]\) (in 2-way nesting), is read and written. Therefore, by applying the transformation both a load and a store are moved inside the innermost loop and since \(a[i][k]\) is only read, a load is moved outside. The difference is minimal, the code of the innermost loop increases by one instruction. Regarding the performance, the instructions that move into and out of the innermost loop are the most relevant.

Any variable with a stride equal zero is exposing temporal locality, which means exactly the same access is referred. Losing this temporal locality, results in an increase of the number of instructions that the processor needs to execute, which can be seen as a negative factor for the global performance. The inverse case can happen. Whenever a stride of a variable is reduced to zero, the code in the innermost loop is reduced and also execution time. For both sample codes, regarding the register reuse no profit is expected. This increase in the number of instruction

---

1 In Table 3.3, \(l_{\text{inner}}\) refers to the loop index at the innermost position.
### 3.2 Loop Permutation

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Destination</th>
<th>Symbol</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000007b4</td>
<td>e1a0e005</td>
<td>MOV</td>
<td>lr, r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007b8</td>
<td>e79ec008</td>
<td>LDR</td>
<td>r12, [lr, r8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007bc</td>
<td>e08a200e</td>
<td>ADD</td>
<td>r2, r10, lr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c0</td>
<td>e3a01000</td>
<td>MOV</td>
<td>r1, #0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c4</td>
<td>e2811091</td>
<td>ADD</td>
<td>r1, r1, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c8</td>
<td>e49234b0</td>
<td>LDR</td>
<td>r3, [r2], #0x4b0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007cc</td>
<td>e3510f4b</td>
<td>CMP</td>
<td>r1, #0x12c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d0</td>
<td>e86cc003</td>
<td>ADD</td>
<td>r12, r12, r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d4</td>
<td>1af00000</td>
<td>BNE</td>
<td>pc-0x18 ; 0x7c4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d8</td>
<td>e2855001</td>
<td>ADD</td>
<td>r5, r5, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007dc</td>
<td>e3550f4b</td>
<td>CMP</td>
<td>r5, #0x12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e0</td>
<td>e78ec008</td>
<td>STR</td>
<td>r12, [lr, r8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e4</td>
<td>e28ee004</td>
<td>ADD</td>
<td>lr, lr, #4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e8</td>
<td>1af00000</td>
<td>BNE</td>
<td>pc-0x38 ; 0x7b8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007ec</td>
<td>e3a033b2</td>
<td>MOV</td>
<td>r3, #0x8000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Destination</th>
<th>Symbol</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000007b4</td>
<td>e1a0e005</td>
<td>MOV</td>
<td>lr, r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007b8</td>
<td>e79ec008</td>
<td>LDR</td>
<td>r12, [lr, r8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007bc</td>
<td>e08a200e</td>
<td>ADD</td>
<td>r2, r10, lr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c0</td>
<td>e3a01000</td>
<td>MOV</td>
<td>r1, #0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c4</td>
<td>e2811091</td>
<td>ADD</td>
<td>r1, r1, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007c8</td>
<td>e49234b0</td>
<td>LDR</td>
<td>r3, [r2], #0x4b0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007cc</td>
<td>e3510f4b</td>
<td>CMP</td>
<td>r1, #0x12c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d0</td>
<td>e86cc003</td>
<td>ADD</td>
<td>r12, r12, r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d4</td>
<td>1af00000</td>
<td>BNE</td>
<td>pc-0x18 ; 0x7c4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007d8</td>
<td>e2855001</td>
<td>ADD</td>
<td>r5, r5, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007dc</td>
<td>e3550f4b</td>
<td>CMP</td>
<td>r5, #0x12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e0</td>
<td>e78ec008</td>
<td>STR</td>
<td>r12, [lr, r8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e4</td>
<td>e28ee004</td>
<td>ADD</td>
<td>lr, lr, #4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007e8</td>
<td>1af00000</td>
<td>BNE</td>
<td>pc-0x38 ; 0x7b8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000007ec</td>
<td>e3a033b2</td>
<td>MOV</td>
<td>r3, #0x8000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3. Comparison between original(a) and transformed(b) assembly code for N = 300.
is constant independently of the trip count. If no other benefits due to loop permutation are found, the final speed up is negative.

Now the effect of the transformation on the cache performance is analyzed. For doing so, first the algorithm exposed by Carr et al [7], is applied to both of the sample programs, in order to determine if the order proposed on the transformed code exposes more data reuse in the innermost loop than the original code. The results obtained are shown in Table 3.5 and Table 3.6.

### Table 3.5. Loop Cost to derive the “best” loop order on 3-way nesting code

<table>
<thead>
<tr>
<th>$l_{inner}$</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[i][j]</td>
<td>$N \times N^2$</td>
<td>$\frac{N}{8} \times N^2$</td>
<td>$1 \times N^2$</td>
</tr>
<tr>
<td>a[i][k]</td>
<td>$N \times N^2$</td>
<td>$1 \times N^2$</td>
<td>$\frac{N}{8} \times N^2$</td>
</tr>
<tr>
<td>b[k][j]</td>
<td>$1 \times N^2$</td>
<td>$\frac{N}{8} \times N^2$</td>
<td>$N \times N^2$</td>
</tr>
</tbody>
</table>

Total: $(1 + 2N)N^2$ (1 + $\frac{2N}{8})N^2$ (1 + $\frac{N}{8})N^2$

### Table 3.6. Loop Cost to derive the “best” loop order on 2-way nesting code

<table>
<thead>
<tr>
<th>$l_{inner}$</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>total[i]</td>
<td>$\frac{2}{8} \times N$</td>
<td>$1 \times N$</td>
</tr>
<tr>
<td>a[j][i]</td>
<td>$\frac{3}{8} \times N$</td>
<td>$N \times N$</td>
</tr>
</tbody>
</table>

Total: $(\frac{32k}{N})N$ (1 + $\frac{N}{8})N$

In 3-way nesting, by arranging the loop indexes in descending order in terms of cache lines accessed, it is obtained as the best order: ikj, being j the innermost loop. For 2-way nesting, the best order would be ji. Since those two orders are the ones used in the transformed codes, from the point of view of this algorithm, they expose the best data locality. Then, according to this model a profit is expected. Nevertheless, it is known that the model gives the upper bound of the possible cache misses, and therefore the maximum of the benefits that might be achieved. Other factors can be analyzed to fully describe the behavior shown in Figure 3.2.

First, there is a break-even-point where the problem size becomes bigger than the capacity of the memory. The number of cache lines for the memory configuration used for the ARM is

\[
\#CL = \frac{C}{L_w} = \frac{32kB}{8\text{ words/word}} = 1K \text{ cache lines.}
\]

In the best cases where the cache conflicts are low, the break-even-point for the transformation done in 3-way nesting is given by

\[
\text{Cache Lines}_k > \text{Cache Lines}_{\text{mem}} \Rightarrow (\frac{N}{8} + N + 1) > 1K \Rightarrow N > 909
\]

A similar result is obtained for 2-way nesting:

\[
\text{Cache Lines}_j > \text{Cache Lines}_{\text{mem}} \Rightarrow (N + 1) > 1K \Rightarrow N > 1023
\]
Due to the assumptions, these values represent only an approximation where the capacity of the cache is exceeded and even with no conflict misses, the benefits are present. However, when considering conflict misses, the benefits can be present along all the tested range of trip count. The approximation can be performed for the memory configuration of the other processors and the results are summarized for further analysis in Table 3.7 and Table 3.8. Hereafter these points where the capacity of the cache is exceeded are called break-even-points, and they are also shown in Figure 3.2 for the 2-way nesting sample code.

Table 3.7. Break-even-point where problem size exceeds cache capacity in 3-way nesting code

<table>
<thead>
<tr>
<th>ARM926EJS</th>
<th>TI-C6455</th>
<th>Diamond 570T</th>
</tr>
</thead>
<tbody>
<tr>
<td>909</td>
<td>454</td>
<td>221</td>
</tr>
</tbody>
</table>

Table 3.8. Break-even-point where problem size exceeds cache capacity in 2-way nesting code

<table>
<thead>
<tr>
<th>ARM926EJS</th>
<th>TI-C6455</th>
<th>Diamond 570T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>511</td>
<td>255</td>
</tr>
</tbody>
</table>

The fluctuations of the curve in Figure 3.2 are the response to the different occurrence levels of the conflict misses. As it was stated by Ghosh [9], conflict misses are highly sensitive to slight variations in problem size and base addresses.

Until now, it has been analyzed the performance behavior of loop permutation for the ARM processor, which has a Scalar RISC architecture that constrains to a completely sequential execution of the code. The analysis has shown important factors that determine the profitability of a transformation when the program runs sequentially on a uniprocessor like ARM. However, the same program was run on TI-C6455 and Diamond 570T, both VLIW architectures, where some level of parallelism is used.

TI-C6455

For the Figure 3.2 the dashed red line corresponds to the behavior of the transformation when applying it on code that is executed on Texas Instruments C6455 processor. Previously, it was shown that the direction matrixes for both 3-way nesting and 2-way nesting, change respectively as follows:

\[
\begin{align*}
( = = < ) & \rightarrow ( = < = ) \\
( < * ) & \rightarrow ( * < )
\end{align*}
\]

The modification of the dependence pattern can be used to determine if the transformation exposes parallelism at a granularity that might be exploited by the specific architecture. Since the TI-C6455 is a machine that exploits the instruction-level parallelism, then it is pursued to move independent loops to the innermost position.
For 3-way nesting, an independent loop was moved to the innermost position, which might expose instruction level parallelism. For 2-way nesting nothing can be said about parallelism by looking at the dependence pattern, since all possibilities are included in the first element of the direction vector. When checking the assembly code it was found that the the ILP is better exploited on the original code.

The C6000 compiler attempts to apply a software-pipelined loop scheduling technique, which allows iterations to be overlapped, i.e. the iteration i+1 can start before iteration i finishes. In a software pipelined loop, a single iteration might take s cycles to complete, however a new iteration is initiated every ii cycles. Formally, ii is called initiation interval and it is the latency between iteration i and iteration i+1, s is called single-scheduled iteration, and is the number of instructions of a single iteration of the software-pipelined loop [10].

The resultant software pipeline schedule, found in the assembly code, shows to be sensible to the order of the nested loop. The collected statistics are shown in Table 3.9.

Table 3.9. Software pipelined scheduling in TI-C6455 when applying loop permutation

<table>
<thead>
<tr>
<th></th>
<th>3-way nesting</th>
<th>2-way nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>ii</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

The table shows that for both cases the number of instructions of a single iteration is smaller for the original code. For the original code of 3-way nesting, the initiation interval is 4, which means that in the steady state, a result is computed every 4 cycles, and it does 4 iterations in parallel, for the transformed version instead 9 cycles are needed to obtained a new result and 3 iterations are done in parallel. This shows that the original code performs better. 2-way nesting has more critical results, for the original code a result is generated every 2 cycles, and it performs 4 iterations in parallel, while the transformed code does only 2 iterations in parallel and a result is delivered every 47 cycles.

Comparing the scheduled instruction sequence for a single iteration between the original and the transformed code, it was seen that the original order of the loop nest favors the selection of wider load instructions, that are used instead of multiple loads to balance and reduce the number of resources. It was also notable the reduction in the use of store operations due to the temporal reuse of the variables that need to be written in both original codes.

According to the schedule generated no profits are expected. The schedule remains the same when varying the trip count. Hence, the benefits achieved in the Figure 3.2 are due to the improvement of the cache performance, that is now analyzed.

For the memory configuration used for this processor, the number of cache lines is 512. Therefore, doing the same calculation as for the ARM, the break-even point where the capacity of the cache is exceeded can be approximated for 3-way nesting to 454 and for 2-way nesting to 511. One can notice that the average speedup is increased around these values for each case.

Additionally, using the profiling tools included in Code Composer it is possible to corroborate that the fluctuations seen in the graphic are related with the occurrence of the conflict misses, as it was stated in the analysis done for the ARM.

The number of sets for the used memory configuration is \( N_s = 256 \), whose prime decomposition is \( 2^8 \). Previously, it was mentioned that the greatest common divisor (gcd) between
3.2 Loop Permutation

the number of sets \( (N_s) \) and the array size in terms of cache lines \( (\lfloor N_s L_s \rfloor) \), is used as a factor of reduction of the number of sets accessed from the total number of sets available. For the value of \( N_s \) of this memory, the reduction factor is \( RF = 2^x \) where \( x \in \{1, 2, \ldots, 8\} \), this is due to the fact that \( \gcd(N_s, \lfloor N_s L_s \rfloor) \leq \min(N_s, \lfloor N_s L_s \rfloor) \).

Looking at values smaller than 454 where capacity misses are not expected in 3-way nesting, among the points sampled, the point where \( N_s = 400 \) shows a lower point in the curve, with exactly a speedup of 1.05. The prime decomposition of this value correspond to \( 2^4 \cdot 5^2 \), since the \( L_s \) is equal to \( 2^4 \) words, then

\[
RF = \gcd(N_s, \lfloor \frac{N_s}{L_s} \rfloor) = \gcd(2^8, 5^2) = 1
\]

Since the reduction factor for this case is one, the total number of sets available are likely to be used when iterating the loop, and a small occurrence of conflict misses is expected. Therefore, when the number of conflict misses is small the transformation is less likely to be beneficial. To corroborate this results, two new contrasting values less than 454 are chosen, and the profiling tool is used to obtain the statistics shown in Table 3.10.

- \( N = 240 = 2^4 \cdot 3 \cdot 5 \Rightarrow RF = \gcd(2^8, \lfloor 2^4 \cdot 3 \cdot 5 \rfloor) = 1 \)
- \( N = 256 = 2^8 \Rightarrow RF = \gcd(2^8, \lfloor 2^8 \rfloor) = 2^4 = 16 \)

Table 3.10. Cache statistics in 3-way nesting when applying loop permutation

<table>
<thead>
<tr>
<th></th>
<th>Non-Conflicting (N=240)</th>
<th></th>
<th>Conflicting (N=256)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Transformed</td>
<td>Original</td>
</tr>
<tr>
<td>Conflict misses</td>
<td>196</td>
<td>0</td>
<td>1129744</td>
</tr>
<tr>
<td>Miss summary</td>
<td>871396</td>
<td>871200</td>
<td>2186512</td>
</tr>
<tr>
<td>Hits</td>
<td>59510</td>
<td>81106</td>
<td>67573</td>
</tr>
<tr>
<td>Accesss</td>
<td>8726400</td>
<td>29404800</td>
<td>10584064</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.95</td>
<td></td>
<td>1.77</td>
</tr>
</tbody>
</table>

The value of \( RF \) is only an heuristic that considers some of the factors that might generate conflict misses. There can be cases where the reduction factor is one and the number of conflicts is still large due to other factors, for instance, the relative positioning between references, which can also generate cache misses.

Some observations from the data obtained are presented. First, the variation in the problem size is small, \( N=240 \) and \( N=256 \) are close to each other, and the speedup in each case is different. For the non-conflicting case, the original version generates only 196 conflict cache misses. Even if the transformation eliminates all conflicts, the total reduction is negligible compared to the conflicting case, where the original code is critical in terms of conflict misses. The transformation favors the reduction to half of the total misses. This contributes to a speedup of 1.77, even larger than the values of the sample points shown in the Figure 3.2.

Tensilica Diamond 570T

In case of the Tensilica processor, for the permutation code, no improvement for any value of \( N \) was found. The assembly code generated for the original and the permuted source code are exactly the same. Therefore, the changes done on the code at a source level do not produce
any effect to the compiler. It is not possible to determine the decisions that the compiler takes when optimizing the code. However, an hypothesis for the behavior is that the compiler applies either loop permutation or an equivalent transformation as part of its high level optimizations. For instance, if the code is passed without transformation, the compiler might perform loop permutation, because a better loop order is inferred. Now, if the code is transformed at the source level, the compiler does not need to permute it, but the assembly code generated is the same. Hence, there is no difference in the performance.

3.2.6 Conclusions for Loop Permutation

Loop permutation is one of the most useful transformations available for the support of parallelization and memory hierarchy management. This transformation has been analyzed by means of two sample codes, in order to identify in each case the factors that influence the behavior of the performance when loop permutation is applied. The following conclusions can be drawn:

- The order of the loops proposed in the transformations evaluated here does not benefit the register reuse, according to the modification of the dependence pattern and the stride analysis done.

- In terms of the benefits on cache performance when the problem size exceeds the capacity of the cache, the loop order that exposes better data locality in the inner loop is more likely to perform better, since the capacity misses generated prevent the leverage of the data reuse exposed in the outer loops.

- There are certain critical cases that can be generated due to the occurrence of the conflict misses, these cases are independent of the capacity of the cache, and are sensible to slight variations of the array size. When these cases are found loop permutation found a space to generate high speedup when an improvement of the data locality of the innermost loop is applied.

3.3 Loop Reversal

3.3.1 Definition

Loop reversal changes the direction in which the loop traverses its iteration range, as follows:

```
Listing 3.13. Reversal Example 1-Original
for (j = 1; j < N; j++)
    for (i = 1; i < N; i++)
        a[i][j] = a[i-1][j+1] + 1;
```

```
Listing 3.14. Reversal Example 1-Reversed
for (j = 1; j < N; j++)
    for (i = N-1; i >= 1; i--)
        a[i][j] = a[i-1][j+1] + 1;
```

It achieves performance enhancement by:

- Reduce loop overhead: by eliminating the need of a compare instruction, in architecture without a compound compare-and-branch. When the iteration variable is reversed so that it runs down to zero, it allows the loop to end with a branch-if-not-equal-to-zero (BNEZ) \[5\].
3.3 Loop Reversal

- **Enabling other transformation:** it is often used in conjunction with other reordering transformations because it changes the dependence vector, specially used with permutation.

For the application of reversal, the only parameter that needs to be known is which loop of the nest needs to be reversed.

3.3.2 Legality Test

The legality test of reversal is associated on how the transformation modifies the direction vector. For reversal, if loop $p$ in a nest of $n$ loops is reversed, then for each dependence vector $D(i)$, the element corresponding to $p$ is negated, which means that “$<$” becomes “$>$” and vice versa. Then, considering the dependence vector

$$D(i) = (d_1, d_2, ..., d_p, ..., d_n)$$

and assuming that the loop $p$ is the one that runs backwards after applying loop reversal, then the dependence vector $D'(i)$ becomes $D'(i)$, where

$$D'(i) = (d_1, d_2, ..., \bar{d}_p, ..., d_n)$$

Again, the reversal is legal if each resulting vector $D'(i)$, or each row in the direction matrix, is lexicographically positive, which means that it does not have as first non-“$=$” component a “$>$” and it guarantees that the source of the dependence is executed first.

3.3.3 Profitability of Reversal

The application of reversal can generate profits in the performance by reducing the loop overhead. It can allow the use of a single instruction at the end of the loop, such as branch-if-not-equal-to-zero (BNEZ), to make compare and branch at the same time. This might lead to a reduction of the assembly code in the innermost loop.

Regarding the memory subsystem, reversal has a minimal impact on data cache locality. By reversing a loop the stride analysis results do not change. Hence, the locality exposed in each level of the nested loop remains the same.

An analysis of the direction matrix shows the opportunities to apply reversal not as profitable transformation but as enabling transformation, which is the main usage of this transformation. Only one row in the direction matrix is changed at a time. This modification might have different effects:

- Enable the application of another transformation. For example, loop permutation has a legality test that in case it fails, loop reversal might modify the code and therefore the dependence pattern such that the legality test of loop permutation succeed afterward.

- Enable parallelism. If the dependence pattern contains only “$>$” as elements of a single row, by reversing this row and interchanging it with an outer position, it is possible to guarantee that the outer loop carries the dependences. This implies that independent loops are moved to inner positions and vectorization is enabled.

To illustrate the first effect, look at the example code of Listing 3.13 and 3.14, it has a anti-dependence whose direction vector changes as follow when applying reversal:
Permutation of the loops $IJ$ was not legal previous to the reversal, since the resulting pattern for permutation would be $(>,$ $<$), which implies a violation of the dependence. After reversal, the permutation is possible, so that the best order in term of data locality is achieved. Basically the conversion of a “$>$” to a “$<$” in a direction vector enables other transformations, such as loop interchange.

### 3.3.4 Source Code for the Evaluation of Reversal

Since reversal is an enabling transformation. The evaluation is done with three different versions: the original (Listing 3.15), only reversed (Listing 3.16) and reversed and permuted (Listing 3.17).

**Listing 3.15. Reversal-Original**

```plaintext
for (j = 1; j < N - 1; j++)
    for (i = 1; i < N - 1; i++)
        a[i][j] = a[i+1][j-1] + 5;
```

**Listing 3.16. Reversal-Reversed**

```plaintext
for (j = 1; j < N - 1; j++)
    for (i = N - 2; i > 0; i--)
        a[i][j] = a[i+1][j-1] + 5;
```

**Listing 3.17. Reversal-Reversed+Permuted**

```plaintext
for (i = N - 2; i > 0; i--)
    for (j = 1; j < N - 1; j++)
        a[i][j] = a[i+1][j-1] + 5;
```

### 3.3.5 Results

The following figure shows the results of the application of loop reversal in the three considered processors. Figure 3.4b presents the effect of reversing only the inner loop. And Figure 3.4b shows the speedup obtained when reversal is used in combination with permutation.

Summary of Program Context Factors

Before the analysis of the behavior of the transformation in each single processor, a summary of the main factors influencing loop interchange according to the previous section is given.

**Dependence pattern:** Looking at the code, there is only one true-dependence. Hence, the direction matrix correspond to a single row:

$$DM(j, i) = \begin{pmatrix} < & > \end{pmatrix}$$

Notice that without loop reversal, the interchange of the loops is illegal. Also the application of loop reversal is only legal on the $i$-loop. By applying reversal the dependence is modified to

$$DM'(j, i) = \begin{pmatrix} < & < \end{pmatrix}$$

The application of loop permutation does not generate any modification in the dependence pattern of this code. Therefore, no benefits regarding register reuse or enabling parallelism are generated. Since no register reuse is lost or won, the number of access to the memory also remains the same.
3.3 Loop Reversal

**Stride Analysis:** Reversing the loop does not modify the stride analysis, but loop permutation does. The modification is shown in Table 3.11.

From the stride analysis is also seen that there is no loss or win of temporal locality for enabling the register reuse.

**Best loop order:** the best loop order based on the data locality expose in terms of cache lines accessed in the innermost loop is obtained by applying the LoopCost algorithm, the results are summarized on Table 3.12.

Based on the algorithm, the best loop order is actually the one applied by the transformed code, i.e., as the innermost loop.

**Break-even-point:** the breakpoint where the capacity of the cache is exceeded, can be approximated by means of the number of cache lines in each memory configuration and the number of caches lines accessed when iterating the innermost loop of the original code, i.e. $trip_i = N - 2$. 

---

![Graph (a)](image_url)

![Graph (b)](image_url)

Figure 3.4. Speedup when applying (a) only reversal and (b) reversal and permutation.
Table 3.11. Stride analysis when applying loop permutation on reversal sample code

<table>
<thead>
<tr>
<th>Variable</th>
<th>Original Stride ($l_{inner} = i$)</th>
<th>Transformed Stride ($l_{inner} = j$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[i][j]$</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>$a[i+1][j-1]$</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.12. Loop Cost to derive the “best” loop order on reversal sample code

<table>
<thead>
<tr>
<th>$l_{inner}$</th>
<th>i</th>
<th>j</th>
<th>$trip_i$</th>
<th>$trip_j$</th>
<th>$L_i$</th>
<th>$L_j$</th>
<th>$L_s$</th>
<th>$trip_s$</th>
<th>$L_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[i][j]$</td>
<td>$trip_i$</td>
<td>$trip_j$</td>
<td>$L_i$</td>
<td>$L_j$</td>
<td>$L_s$</td>
<td>$trip_s$</td>
<td>$L_s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a[i + 1][j - 1]$</td>
<td>$trip_{i+1}$</td>
<td>$trip_{j-1}$</td>
<td>$L_{i+1}$</td>
<td>$L_{j-1}$</td>
<td>$L_s$</td>
<td>$trip_{i+1}L_{i+1}$</td>
<td>$L_s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>$trip_i + trip_j$</td>
<td>$2 \cdot trip_s$</td>
<td>$L_i$</td>
<td>$L_j$</td>
<td>$L_s$</td>
<td>$2 \cdot trip_sL_s$</td>
<td>$L_s$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ARM926EJS

Figure 3.4.a shows that the application of only loop reversal does not generate any significant speedup in the performance of the code. This processor does not count with a BNEZ instructions to achieve the possible benefit of overhead reduction. In this case, loop reversal serves only as enabling transformation for permutation.

From the benefits related with loop permutation, register reuse does not apply for the program considered. Therefore, the enhancement of the cache performance is the only benefit that generates a speedup. The same behavior analyzed in the previous section is also obtained with this sample code, for example the fluctuation of the curve due to the conflict misses. It was sampled 3 conflicting cases at 128, 256 and 800 which are the highest speed ups seen in the graphic. After the break-even-point that was calculated on 961, positive speedups were also present.

TI-C6455

For the C6455 processor, applying loop reversal on its own does not provide any speedup neither. However, for the case where it is combined with loop permutation it does. Due to the characteristics of the memory, the break-even-point where the capacity of the cache is exceed is visualized in the sampled range and it is depicted in the Figure 3.4 as vertical lines. After this break-even-point, while the trip count increases, so does the global speedup, until the upper bound in the cache misses is found that is when the data reuse exposed in the outer loop is lost.

Tensilica Diamond 570T

For Tensilica Diamond 570T, the same behavior is seen in both versions of the code, therefore no extra benefits are added due to the permutation of the loops. This was expected since in the previous analysis loop interchange did not show any profit on this processor. Once the break-even-point is passed, the speedup increases until values between 8.22 and 9.21, which represent great improvements. Notice that the behavior seen in this results matches with the hypothesis stated when analyzing loop permutation. Due to the original dependence pattern,
applied loop permutation would be illegal to the compiler. When only loop reversal is applied at the source level, loop permutation become legal and the compiler is allowed to apply it among its high-level optimizations. This is the reason why the application at the source level of both, loop reversal and loop permutation, does not show any difference.

3.3.6 Conclusions for Loop Reversal

The main conclusions obtained from the evaluation of loop reversal as source to source transformations are the followings:

- No additional benefits were found for the global performance due to the application of only loop reversal. This was explicitly corroborate for the ARM926EJS and TI-C6455. It is also presumed that the benefits seen at Tensilica Diamond 570T are actually due to the application of loop permutation by the compiler.
- The main objective of this transformation is to enable permutation when it is illegal to be applied, due to the dependence pattern. The application of permutation for this sample code showed important improvements of the performance that rises to values greater than 8 times the original execution time. Without loop reversal this code is not able to get any benefit.

3.4 Loop Skewing

3.4.1 Definition

Loop Skewing is a transformation that reshapes the iteration space. It is performed by adding the outer loop index multiplied by a skew factor $f$, to the upper and lower bound of the inner loop variable, and then subtracting the same quantity (outer loop variable * $f$), from every use of the inner loop variable inside the loop. This procedure is illustrated in the following example where the inner loop($j$) is skewed with respect to ($i$), using $f=1$.

<table>
<thead>
<tr>
<th>Listing 3.18. Skewing Example 1-Original</th>
<th>Listing 3.19. Skewing Example 1-Skewed</th>
</tr>
</thead>
<tbody>
<tr>
<td>for($i = 1; i &lt; N; i++)</td>
<td>for($i = 1; i &lt; N; i++)</td>
</tr>
<tr>
<td>for($j = 1; j &lt; N; j++)</td>
<td>for($j = i + 1; j &lt; i + N; j++)</td>
</tr>
<tr>
<td>$a[i][j] = a[i-1][j] + a[i][j-1]$;</td>
<td>$a[i][j-1] = a[i-1][j-1] + a[i][j-1]$;</td>
</tr>
</tbody>
</table>

The transformed code is equivalent to the original code. The effect of the transformation on the iteration space is illustrated in Figure 3.5. The idea behind this reshaping is to expose the existing parallelism in a conventional loop. In Figure 3.5 both of the axes contain dependences. Therefore, none of the loops are parallelizable. However, the updates in the array propagates as waves across the iteration space, creating diagonal lines of parallelism also called wavefronts.
By reshaping the iteration space, it is possible to reveal the existing parallelism of the code in a conventional parallel loop.

The transformation can be also seen as a remapping. For example, to skew $j$ with respect to $i$, a new index variable $jj$ is created where $jj = j + i$. Using $jj$ to replace $j$ given the inverse mapping $j = jj - i$. The resultant code is the same as the one previously presented.

Skewing can generate performance enhancement when enabling the following items:

- **Parallelism and vectorization**: by converting ‘=’ to ‘<’ in a direction vector.
- **Other transformations**: Just as reversal, it is often used in conjunction with other reordering transformations since it produces changes in the dependence vector. Other transformations that were at first place illegal can be applied [5], for example permutation.

For applying skewing the parameters needed are: the skew factor, and the loop that wants to be skewed within a loop nest. The wrong selection of a skew factor might reduce the maximum profit that can be achieved. However, its selection is not difficult when the dependence pattern of the program is known.

### 3.4.2 Legality Test

This transformation modifies the bounds, but then it also changes the uses of the corresponding index variables to compensate this modification. Since skewing does not change the meaning of the program, it is always legal. Although this transformation has been treated as loop reordering transformation, it merely relabels the iteration space, no computation reordering is done.

### 3.4.3 Profitability of Skewing

Skewing is a transformation of the iteration space, and this implies also a modification of the dependence direction matrix. To illustrate how the dependence direction matrix is modified consider the code shown in Listings 3.20 and 3.21.
In a direction matrix, the direction vectors affected by the changes in the iteration space are those that carry dependences by the loops used as reference for skewing a loop. In the example, the innermost loop is skewed with respect to the two other loops by using the substitution $kk = k + i + j$. The original and transformed direction matrix are:

$$
\begin{bmatrix}
= & < & = \\
< & = & < \\
= & < & < \\
= & = & = \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
= & < & < \\
< & = & < \\
= & < & < \\
= & = & = \\
\end{bmatrix}
$$

If only the inner loop is skewed with respect to the outer loop, then the direction vector affected is only the second one, which carries the dependence in the outer loop. This leaves the following transformed direction matrix:

$$
\begin{bmatrix}
= & < & = \\
< & = & < \\
= & < & < \\
= & = & = \\
\end{bmatrix}
$$

For the transformation example shown on Listings 3.18 and 3.19, the direction vector was modified as follows:

$$
\begin{bmatrix}
= & < \\
< & < \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
= & < \\
< & < \\
\end{bmatrix}
$$

This modification is also graphically illustrated in Figure 3.5.

Now, considering how the direction matrix is modified, the profitability of skewing relies in two possible uses. One, to transform the skewed loop into one that can be interchanged to an outer position without violating any dependence, i.e. as enabling transformation. Second, an inner loop can be skewed so that a column containing only “<” is generated. Subsequently, the interchange of this column to the outermost position exposes parallelism.

Notice that loop interchange is always a good option to be applied with enabling transformations like this one, due to its benefits when enabling parallelism or enhancing locality. The code in Listing 3.22 shows the result of applying loop permutation to the skewed code in Listing 3.19.
3.4 Loop Skewing

Listing 3.22. Skewing Example 1-Skewed + Permuted

```c
for(j = 2; j <= N+N; j++)
  for(i = max(1,j-N); i <= min(N,j-1); i++)
    a[i][j-i] = a[i-1][j-i] + a[i][j-i-1];
```

Note that the inner bounds depend on the iteration variables of the enclosing loops, this makes the application of loop permutation on skewed loops difficult. Also, when applying loop permutation, the number of iterations of the outer loop is increased to $2 \times \text{tripcount} - 1$, and the number of iterations of the inner loop is varying from 1 to N. Therefore, the application of the LoopCost algorithm is not possible. Nevertheless, an enhancement in locality might be achieved depending on how the array is traversed after the transformation. The goal of this transformation, rather than the enhancement of locality is the enabling of the parallelism shown as wavefronts.

Regarding the selection of the skew factor, it is desired to choose the minimum value that exposes the parallelism or enables other transformations. Once this minimum factor is found, to increase the value reduces the amount of iterations that can be done in parallel and therefore it always implies less profit. The effect of the skewing factor is only reflected in the dependence distance.

To convert an inner loop into a dependence-free loop for example, the inner loop is skewed with respect to the outer loop, such that the distances of the dependences carried by the outer loop are at least 1. This might imply a large factor if negative distance are found in the dependence distance vector.

Applying skewing has a non-zero cost and several drawbacks can be mentioned. The loops that result from this transformation are highly irregular (trapezoidal shape), which implies load imbalances in the use of resources. Also, since the inner bounds depend on the outer loop, they need to be recomputed for each iteration of the outer loop.

3.4.4 Source Code for the Evaluation of Skewing

The sample code used for the evaluation of skewing is shown in Listings 3.23 and 3.24. Since it is mainly considered as an enabling transformation, the transformed code presented is the one consisting of both transformations, skewing and permutation. Nevertheless, both versions are measured, the skewed-only and skewed-permuted. The results obtained are presented in the next Section.

Note that the original version of the source code, in terms of the references accessed, is the same as the original source code used in the evaluation of loop reversal. Therefore, the dependence pattern and the break-even-point approximations for each memory configuration are valid for the analysis of skewing. The strides when applying only skewing are not modified since there is no reordering of the memory accesses. The data locality exposed in the inner loop cannot be analyzed by the LoopCost algorithm because the trip count is varying, and in each iteration of the outer loop a different amount of cache lines accessed is expected.

Listing 3.23. Skew-Original

```c
for(j = 1; j < N - 1; j++)
  for(i = 1; i < N - 1; i++)
    a[i][j] = a[i+1][j-1] + 5;
```
### 3.4.5 Results

Figure 3.6 shows the results of the application of skewing in the three considered processors. Figure (a) presents the effect of applying only skewing to the inner loop. And Figure (b) shows the speedup obtained when using skewing in combination with permutation.

**ARM926EJS**

Once again, in case of the ARM no parallelism can be exploited. Therefore, the benefits that might be achieved are regarding the memory subsystem. First, the application of skewing with a factor equal to 1 modifies the dependence pattern as follows:

\[( < > ) \rightarrow ( < = )\]

Note that to determine the minimum skewing factor to enable permutation, the exact value of the distance vector is needed. In the sample code, this is \((1, -1)\). Then, skewing by a factor of 1, generates a valid direction vector for the application of loop permutation, which previously was illegal.

The application of only skewing for this processor does not generate any profit. The exposed parallelism cannot be leveraged, and in terms of memory accesses no modification or reordering is done by only skewing the loop.

From the dependence pattern, it can be seen that the interchange of the loops enables register reuse, giving a positive profit for the permutation, when considering only the register level. Nevertheless, this contribution is reduced by the cost of applying skewing since the bounds need to be recomputed. Minimum and maximum functions are used in each iteration of the outer loop, adding more instructions to the execution of the innermost loop. This can be observed in Figure 3.6 (b). The obtained profit is negative for smaller values of the trip count where no conflict misses are expected.

For analyzing the effect on the cache performance, a data locality analysis is needed. The way the data locality is affected, when applying both skewing and permutation, has not been formally described or modeled in literature. However, by inspecting the way the array is traversed for this particular sample code, several facts are identified:

- The stride is not reduced to 1 even when the inner loop variable is used to reference the rightmost dimension, due to the skewing transformation, it is also used in the leftmost dimension. Therefore, after the permutation, the stride remains the same, i.e. equals to N.
The way the array is traversed changes from column to diagonals. Then, the number of iterations between accesses to contiguous elements in memory increases in each iteration of the outer loop, reducing the possibility to realize the spatial reuse that is usually gained when the stride is reduced to 1 by loop permutation.

According to the last statement, the first and the last iterations of the outer loop guarantees the realization of the spatial reuse, but when the trip count increases these benefits become negligible and the gain tends to vanish.

From previous sections, it was seen that loop interchange usually looks for exposing parallelism, or to enhance locality. When it is applied in combination with skewing the behavior observed is different, since the upper bound in cache misses is not constant. For the iteration range sampled, this can be observed for the memory configurations of the next two processors.
TI-C6455

For this processor, if the loop is only skewed no improvement is found. Additionally, when checking the generated assembly, the solution given by the compiler to the software pipeline scheduling is the same for both the original and the transformed code. The number of instructions and the initiation interval values for each case are shown in Table 3.14.

Table 3.14. Software pipelined scheduling when applying skewing and permutation

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>OnlySkewed</th>
<th>Skewed and Permuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>8</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>ii</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

When applying both, permutation and skewing, the solution of the software pipelining is different. The number of instructions for a single iteration and the initiation interval increase. Other important information is that due to the recalculation of the bounds the compiler cannot infer the maximum trip count. Consequently, the loop carried dependences bound reported are 7, and the use of the resources is unbalanced.

The spatial reuse realized by the permutation of the skewed code, is less than the reversed and permuted case for example, where the stride is reduced to one. Therefore the maximum speedup obtained is less for this case, around 2 times. As stated before in the analysis for the ARM, the difference between the spatial reuse realized by the skewed and the original code does not depend on the trip count. Therefore, when increasing N, the benefits become less important and the global performance is reduced.

Tensilica Diamond 570T

If only skewing is applied, the dependence pattern is modified and loop permutation is enabled. Previously in the analysis done for loop reversal and loop permutation, it was presumed that the compiler might be applying loop permutation among its high-level optimizations. However, in this transformation the behavior seen when applying the permutation at source level is different from the one obtained by only skewing the loop. How the compiler does its optimization is unknown at this level, but the modification in the dependence pattern is enabling a transformation within the compiler. Also, note that the maximum speedup achieved for this processor has also been reduced compared with the one achieved in the reversed case.

When applying both skewing and permutation, as for the TI processor, when increasing the trip count, the spatial reuse gain is important at some point, then the global performance is reduced and it tends to 1.

3.4.6 Conclusions for Loop Skewing

The main conclusions for these transformations are the following:

- Its main usage is an enabling transformation either for the application of other transformation, or for exposing the existing parallelism in the loop in a conventional loop.
- Different to loop reversal, the application of skewing has a cost, mainly due to the re-computation of the bounds.
• The transformation combined with loop permutation in order to enhance locality is inefficient when increasing the trip count value. The way skewing is performed avoids the reduction of the stride to one, and even if the array is traversed differently, the data reuse realized is important only for some values of the trip count. Therefore, when increasing the trip count of the loop the global performance obtained tends to 1.

Regarding each processor, the following facts were observed:
• By its own, no profits were obtained for any of the processor here evaluated.
• For the ARM926EJS, the enhancement in data locality does not show high speedups, due to the characteristics of the memory, for the values of the trip count where the capacity is exceeded the benefits of the realized data reuse are already neglectible.
• For the TI-C6455 processor, the permutation of the skewed code, prevents the compiler from calculating the maximum trip count. Then, it assumes a conservative approach regarding the loop dependence bound, and unbalances in the resources are seen.
• For the Tensilica core, the application of only skewing in the loop, enables a transformation within the compiler. The maximum speedup achieved is smaller than the one obtained when the compiler transformation is enabled by means of reversal for the same sample code.

3.5 Tiling
3.5.1 Definition
Also called blocking, tiling is a combination of strip-mining of inner loops with loop permutation. Stripmining converts a single loop into two nested loops for a specified block size “B”. The original loop executes B iterations starting at each value of the new control loop variable. Afterwards, the control loop is interchanged to the outermost position. The example shown in Listing 3.25 and Listing 3.26 illustrates the transformation, where both of the loops i and j are tiled.

Listing 3.25. Tiling Example 1-Original
```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        a[i][j] = b[j][i];
```

Listing 3.26. Tiling Example 1-Tiled
```
for (ii = 0; ii < N; ii += B)
    for (jj = 0; jj < N; jj += B)
        for (i = ii; i < min(ii+B,N); i++)
            for (j = jj; j < min(jj+B,N); j++)
                a[i][j] = b[j][i];
```

It achieves performance enhancement by:
• Improving cache reusage: tiling enhances locality because fewer data elements are accessed between uses of the same data,
• Enhancing the granularity of parallelism: it permutes parallel control loops to the outermost level.

In the case of tiling the parameters needed for the application of the transformation are the block size factor, the order of the loops and which ones to tile. The variation of each of these parameters might generate changes in the profitability of each transformation.
3.5.2 Legality Test

Tiling can be considered as a compound transformation. The legality test is passed if both strip-mining and loop permutation are legal. Stripmining is always legal since it does not change the execution order. It only increases the depth of loop nest. Loop permutation instead, may violate dependences.

If any transformed dependence vector after permutation is not lexicographically positive, then tiling is not legal. Since permutation moves control loops to the outermost level, tiling is legal only if strip-mining does not introduce negative elements into the transformed dependence vectors. In general, a pair of adjacent loops can be tiled if they can legally be interchanged. It is common to use skewing to enable permutation.

3.5.3 Profitability of Tiling

Loop permutation selects the best loop ordering considering locality, and further improvement can be achieved by blocking. When analyzing loop permutation it was seen that the realization of the data reuse exposed in the outer loops, depends on the size of the arrays. If a single column or row of an array does not fit in cache, then no data reuse of the outer loops can be realized. By blocking the rows and fixing the columns to a size equal to the blocking factor \( B \), the full realization of the data reuse is guaranteed.

However if the referenced array is not aligned, then blocking has a cost. An array is aligned if each column and row begins on a new cache line. When considering unaligned data where a row might begin late in a cache line and end early in the last cache line, then the inner loop over contiguous sections of the array may incur in additional misses.

In summary, the profitability of blocking depends on the reuse present in the outer loops, which leads to a reduction of cache misses when the transformations is applied, and it also depends on the cost of the blocking which means the number of extra misses introduced. The best order inferred if only loop permutation is applied, is not necessarily the best order if also blocking is performed. For example, it may pay to have a constant access instead of a contiguous access because this might move the extra misses out of the inner loop.

In [14] it is stated that the reduction of cache misses due to the realization of the data reuse in outer loops is proportional to the trip count \( N \) of the loop and the extra misses are approximately \( M/B \) for each independent reference in the inner loop, where \( M \) is the trip count of the inner loop and \( B \) the blocking factor.

3.5.4 Source Code for the Evaluation of Tiling

The sample code used for the evaluation of tiling is shown in Listings 3.27 and 3.28. It consists of the same matrix multiplication code used in loop permutation.

**Listing 3.27. Tiling-Original**

```plaintext
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

**Listing 3.28. Tiling-Tiled**

```plaintext
for (kk = 0; kk < N; kk += B)
  for (jj = 0; jj < N; jj += B)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk + B, N); k++)
        for (j = jj; j < min(jj + B, N); j++)
          c[i][j] += a[i][k] * b[k][j];
```
Since the original sample code is the same as the 3-way nesting code used in loop permutation, then the break-even points for that case are still valid for this analysis, and they were summarized in Table 3.7.

### 3.5.5 Results

For each considered processor, the transformation was applied to the sample code using three different values of blocking factor: 32, 64 and 160, and with variations of the trip count (N).

![Figure 3.7. Speedup when applying tiling on the ARM926EJS](image)

In Figure 3.7, the speedup in performance when applying tiling on the ARM926EJS is presented. With loop permutation analysis it was seen that the loop order that exposes the greater locality is \( ikj \), where \( j \) is the innermost loop. A summary of the data reuse expose in each loop is presented in Table 3.15.

### Table 3.15. Summary of the data reuse in each loop for the matrix multiplication code

<table>
<thead>
<tr>
<th>loop</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>a[i][k]</td>
<td>No reuse</td>
<td>Temporal</td>
</tr>
<tr>
<td>c[i][j]</td>
<td>No reuse</td>
<td>Spatial</td>
<td>Temporal</td>
</tr>
<tr>
<td>b[k][j]</td>
<td>Temporal</td>
<td>Spatial</td>
<td>No reuse</td>
</tr>
</tbody>
</table>

After \( j \) is chosen, the next innermost is \( k \). It exposes reuse due to \( c[i][j] \) and \( a[i][k] \). The outermost loop \( i \) also exposes a small amount due to \( b[k][j] \). This data reuse is realized when applying tiling. A similar analysis to the one done with the \textit{LoopCost} algorithm (see Section 2.7.1), is presented in Table 3.16 to show the upper bound of cache misses when blocking is also applied. This analysis considers the extra misses introduced due to the blocking technique.
Table 3.16. Cache misses exposed in each loop for the matrix multiplication code

<table>
<thead>
<tr>
<th>loop</th>
<th>j</th>
<th>k</th>
<th>i</th>
<th>jj</th>
<th>kk</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[i][j]</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{BN}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
</tr>
<tr>
<td>a[i][k]</td>
<td>$1$</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{BN}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
</tr>
<tr>
<td>b[k][j]</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{B}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
<td>$\frac{N^2}{L_s}$</td>
</tr>
<tr>
<td>Total</td>
<td>$\frac{2N^2}{BL_s} + \frac{N^2}{L_s}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the inner loop j, the array c and b expose spatial locality. The number of misses is $B/L_s$, where B corresponds to the length of the rows in a tile and $L_s$ to the cache line size. For the array a only one miss is obtained due to the temporal reuse regarding the innermost loop.

Then, for the next loop k, the number of misses for c is multiplied by 1 since it exposes temporal reuse. The number of misses for a is multiplied by B, since it exposes no reuse, and due to the spatial reuse of b its number of misses is multiplied by $B/L_s$. Doing a similar analysis of the exposed reuse of each referenced array, for the loop i, it is multiplied the number of misses for c and b by N and the number of misses for a by 1. Each of the two outer loops, kk and jj, multiply miss counts by $N/B$, since there is one extra miss per tile.

The original version, which suffered approximately of $N^2(\frac{N}{L_s} + N + 1)$ misses, has an improvement over the total number of cache misses ($\frac{N^2}{L_s}(\frac{2N}{B} + 1)$) by a factor around $\frac{BL_s+1}{2}$. Additionally, this is the best configuration of the loop nest, where B is such that it is multiple of the cache line size and the block $BxB$ can be completely hold \[14\].

Figure 3.7 shows that the speedup is sensible to the blocking factor B. According to the previous analysis the improvement in the number of misses is proportional to the blocking factor B. Therefore, in principle the maximum value, that also guarantees that the tile $BxB$ fits in the cache, will show better profitability. For the characteristics of memory configured for the ARM and the values of blocking factor tested, this is satisfied. See that using a blocking factor of 64 the tile size is of 16 kB, which still fits in the cache of 32 kB. But when increasing to 160, then the tile size is of 100kB and it does not longer fit. Consequently, the improvements are less.

TI-C6455

In Figure 3.8 is presented the speedup in performance when applying tiling on the TI-C6455.

There is no profit when tiling is applied at source level in the C6455 processor. The behavior seen is not the one expected since the loop order chosen by the transformation theoretically expose better locality than the original. Therefore, when no capacity or conflict misses are present, no speedup is expected, but no a degradation of the performance. It was found that instead the conflict misses were increased. For example in the case of $N=100$ with a blocking factor of 64, the conflict misses encountered are shown in Table 3.17.

It is not clear the factors that influence this degradation and it is required other test cases, with different loop orders and number of referenced arrays in order to identified them.
In Figure 3.8 is presented the speedup in performance when applying tiling on the Tensilica Diamond 570T processor.

For tiling also holds the principle that, if the capacity of the cache is exceeded, then the realization of the data reuse in the outer loops in the original code is lost. Thus, only then enhancing of locality achieved in the transformed code represents improvements in the global performance constantly. The break-even-point for the matrix multiplication code and the memory characteristics configured for simulating on the Diamond 570T was approximated to 221, which is also seen on the Figure 3.9.

Regarding the blocking factor, the cache size is smaller in this case, therefore it is possible to see that a blocking factor of 64 which means a tile size of 16 kB is already reaching the size of the cache. Therefore, the blocking factor of 32 is more likely to perform better. Since loop permutation does not show any improvement in the performance when applied at the source level. The improvements shown by the Figure 3.8 they also represents the improvements that can be achieved over the benefits of loop permutation. Since the upper bound in cache misses is smaller when blocking is applied. Comparing Table 3.16 and Table 3.5, it is possible to see that the reduction factor is proportional to $B$. 

![Figure 3.8. Speedup when applying tiling on the TI-C6455](image-url)
3.6 Summary

By manually applying each transformation it was shown that the profitability changes when specific factors are varying, for example, some resource characteristics, or features of program context.

According to the objectives in this chapter the following conclusions can be drawn:

1. The loop transformations analyzed in this chapter showed improvements in performance if they are applied at source level for the investigated embedded processor’s compilers.
Therefore, it was seen that the application of the transformations at this level makes sense and hence their integration within a tool worth it.

2. Good values of speedup were found, up to 9.15 when applying loop reversal in combination with loop permutation in the Diamond 570T processor with a trip count value of 700. But also cases where the performance was degraded. For instance, in TI-C6455 processor using the same transformation, reversal and permutation, it was obtained an speedup of 0.51 with a trip count value of 400.

3. Even when the transformation enhance the locality of the memories, it was corroborate that the profitability also depends on other factors. When varying the iteration space size, the resultant speedup is changing which means that it is necessary to selectively apply the transformations, only those that are beneficial.

4. The general factors were analyzed and summarized on Table 3.18.

Another desired analysis is to determine the parameters or configurations of each transformations, and to determine if the variation of these configurations show different levels of profit.
Table 3.18. Summary of factors influencing the profitability

<table>
<thead>
<tr>
<th>Influence on</th>
<th>General Factors</th>
<th>Program Context Factors</th>
<th>Resource Factors</th>
</tr>
</thead>
</table>
| Cache Performance     | The increment of the data reuse exploited with the “new” loop nest configuration, which is equivalent to the reduction in the number of cache lines accessed.  
                        | The analysis of capacity misses determines the “break-even-point”, where positive profits are more likely to occur.                                                                                      | Trip count of each loop, Step size                                                                 | Cache line size                                                                                     |
|                       | Conflict misses generate variations in the profitability for slight variation of the iteration space size.                                                                                                   | Data reuse analysis                                                                        | Cache line size, cache size                                                                         |
| Register Reuse        | Using a stride analysis, if after the application of the transformation a stride equal zero is obtained, then it is likely to have positive profit.                                                           | Relative addressing among references, trip count                                            | Associativity, number of cache sets                                                                 |
| Enabling Parallelism  | The movement of the dependences in or out within the nested loop enables parallelism.                                                                                                                      | Direction Matrix                                                                          | Architecture of the target machine                                                                |
|                       | The modification of the dependence pattern in a fixed and known way, can enable other locality-enhancing transformation.                                                                                 | Distance and direction vector                                                               | Architecture of the target machine                                                                |
| Enabling Transformations| It was also found that a modification of the configuration of the loop nest can influence the instruction scheduling solution given by the compiler, for example due to the modification of the number of instructions in the inner loop. | Number of load and store instructions                                                         | Architecture of the target machine                                                                |
| Others                |                                                                                                                                                                                                             |                                                                                            |                                                                                                     |
Chapter 4

Case Study: C-to-C Compiler Extended for Loop Reversal

In the previous chapter, good speedups were obtained when applying the loop reordering transformations at source level. Therefore, the application of these transformations as source to source transformations is still valid for improving the input program for modern embedded compilers. However, the sample codes, used in the previous evaluation, are synthetic code for which the applicability of the transformations is ensured. The transformations need to be evaluated in real world test cases in order to determine, first, if the transformations are applicable in known benchmarks code and if the good speedups are still obtained.

A C-to-C compiler named Trafocc was extended as a case study to evaluate the applicability of the transformations. Loop reversal was selected as the transformation for the proof of concept since it showed the highest speedup. Trafocc is a mirtoc based CoSy compiler. The source to source transformations, that are currently offered, have been implemented as engines. The extension of the compiler consists of the implementation of a new optimization engine called loopreversal.

This chapter gives a short description of the steps followed for reversing any loop. It details how to check the legality and to ensure the correct application of the transformation. An analysis about the possible loops that can be reversed is presented, and results obtained when testing and verifying the engine against a benchmark suite are discussed.

4.1 Application of Loop Reversal

Loop reversal changes the direction in which the loop traverses its iteration range. For a single loop, the iteration range is defined by the following elements:

- Initial value: the first value of the loop variable, represented in CoSy as InitExpr.
- Direction: it is positive \( \text{sgn}(\text{dir}) = +1 \) if the loop variable is incremented after each iteration, and negative \( \text{sgn}(\text{dir}) = -1 \) if it is decremented. Note that when the direction is positive the relation \( \rho \) used in the testing condition(TestExpr) is either \(<\) or \(\leq\), for the negative direction \(>\) or \(\geq\). The loops that have a relation different from these four are not considered in this implementation for application of loop reversal.
• Step size: it is the value subtracted or added to the loop variable. It is contained in the `UpdateExpr`.

• Number of iterations: in the CoSy environment, it is called `IterCount`. It is given by the formula:

\[
IterCount = \left\lfloor \frac{(TestValue - (1 * BNI)) - InitValue}{stepsize} \right\rfloor + (1 * sgn(dir))
\] (4.1)

where `TestValue` is the bound of loop, and the variable `bound not included` (BNI) can take the following values:

\[
BNI = \begin{cases} 
1, & sgn(dir) = +1 \land (\rho) = "<" \\
-1, & sgn(dir) = -1 \land (\rho) = ">" \\
0, & otherwise
\end{cases}
\]

In order to reverse the direction in which the iteration range is traversed, it is necessary to define a new initial value, a new test condition and a new update expression. They are calculated as follows:

• `InitValue_{New} = InitValue + (IterCount - 1 * sgn(dir)) * stepsize`

• `TestValue_{New} = InitialValue`

• `\rho_{new} = \begin{cases} 
\geq, & sgn(dir) = +1 \\
\leq, & sgn(dir) = -1
\end{cases}
```

• `UpdateValue_{New} = -stepsize`

If the four elements are known, then the new values can be calculated. Thus, loop reversal is said to be applicable, when original and generated values are available, might be the case that one or several are unknown at compile time. To perform loop reversal not only the applicability, but also the legality needs to be checked.

### 4.1.1 Loop Reversal Example

In order to illustrate the algorithm, the Listings 4.1 and 4.2 present a simple example for the application of loop reversal. The identification of the required elements in the original code and the generation of the new values for reversing the loop with index `i` is summarized in Table 4.1.

<table>
<thead>
<tr>
<th>Listing 4.1. Loop Case 1 - Original</th>
<th>Listing 4.2. Loop Case 1 - Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(j = 1; j &lt; N; j++)</code></td>
<td><code>for(j = 1; j &lt; N; j++)</code></td>
</tr>
<tr>
<td><code>for(i = 3; i &lt; 12; i=i+2)</code></td>
<td><code>for(i = 11; i &gt;= 3; i=i-2)</code></td>
</tr>
<tr>
<td><code>a[i][j] = a[i-1][j+1] + 1;</code></td>
<td><code>a[i][j] = a[i-1][j+1] + 1;</code></td>
</tr>
</tbody>
</table>

How the four elements, the `InitValue`, the `stepsize`, the direction and the `IterCount`, define the iteration range on the original and the transformed cases is depicted in Figure 4.1. Note that the result of the `IterCount` computation is negative when the direction of the iteration range is negative which means `sgn(dir) = -1`. 
Table 4.1. Summary of loop reversal

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Value</td>
<td>3</td>
<td>3 + (5 - 1) * 2 = 11</td>
</tr>
<tr>
<td>Test Value</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>$\rho$</td>
<td>&lt;</td>
<td>$\geq$</td>
</tr>
<tr>
<td>stepsizes</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sgn(dir)</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>BNI</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>IterCount</td>
<td>$\lfloor \frac{(12 - 1) - 3}{2} \rfloor + 1 = 5$</td>
<td>$\lfloor \frac{3 - 11}{2} \rfloor - 1 = -5$</td>
</tr>
</tbody>
</table>

Figure 4.1. Iteration range of the Loop Case 1 (a) original and (b) transformed

4.2 Implementation Details: Loop Reversal

4.2.1 Loop Requirements

There are many ways in which a loop can be written. The Listings 4.3 and 4.4 show the same code written in two different ways

Listing 4.3. Loop Case 2 - Version 1

```c
int *p = &a[0];
int *q = &b[0];
for (j = 1; j < N; j++) {
    total += *p++ * *q++;
    q = q + 2;
}
```

Listing 4.4. Loop Case 2 - Version 2

```c
int p = 0;
int q = 0;
for (j = 1; j < N; j++) {
    total += a[p++] * b[q];
    q = q + 2;
}
```

In both cases p and q are induction variables of the loop. However, in Listing 4.3 they have pointer types, while in Listing 4.4 they have integer types. Furthermore, in the version
more than one statement increases the induction variable $q$. Other factors like conditional statements within the loop or a non rectangular iteration space can lead to complex cases. To guarantee the correctness of the transformation, the engine must identify the characteristics of the loops to which loop reversal can be safely applied, hereafter called the loop requirements.

The loop to be reversed is analyzed according to its loop marker. A loop marker has associated a list of loop variables, which have been classified in 6 different operators according to their usage in the structure of the loop (see Subsection 2.6.1). Not all operators provide the elements previously mentioned in order to apply the transformation. Therefore, by identifying the loop variables related to a loop, it is possible to filter cases where the transformation can be applied.

It is not possible to reverse any loop that contains either a $\text{mirLoopUpdateVar}$ or a $\text{mirHWLoopCounter}$ loop variable, since the information attached to these kind of loop variables is incomplete. For instance, for $\text{mirLoopUpdateVar}$ the exit condition is unknown.

In case of the $\text{mirIterCounter}$, all the elements needed are explicitly annotated (see Listing 2.12). The initial value, for example, is contained in the $\text{InitExpr}$ annotation. According to the definition, $\text{InitExpr}$ has the form “$v = e$”, where $e$ can be any expression corresponding to the initial value of the loop index. Also, the direction is supplied by means of the $\text{Upwards}$ flag, and the number of iterations is annotated as $\text{IterCount}$. The update statement is of the form “$v = v \pm 1$”. Therefore, the step size is fixed to 1. Additionally, with the relation ($\rho$) stored in the $\text{TestExpr}$ annotation, the BNI can be also determined.

For the $\text{mirLoopControlVar}$ (Listing 2.12), also all the elements can be obtained. The initial value, again is contained in the $\text{InitExpr}$ annotation with the form “$v = e$”. Then, the $\text{UpdateExpr}$ annotation is of the form “$v = v \pm i$”, where $i$ is an arbitrary loop-invariant expression that represents the step size needed for the application of loop reversal. Its sign, either plus or minus, determines the direction of the iteration range. The $\text{TestExpr}$ annotation is also available in this kind of loop variable. Thus, the value of BNI is determined using the relation ($\rho$) contained within this $\text{TestExpr}$ annotation. The iteration counter can be calculated using the equation 4.1. In order to compute it, it is necessary that the $\text{Test Value}$ or bound of the loop (left hand side expression within the $\text{TestExpr}$ annotation) is known and invariant. This represents an additional constraint for this kind of loop variables.

The $\text{mirBasicIndVar}$ annotates the information regarding the induction variables of a loop. According to the definition of induction variables, there is no loop with only induction variables. In a loop, it is necessary to have a test condition that is not carried by the induction variables, but instead it is carried by what is called here the main loop variable. This main loop variable can either be a $\text{mirIterCounter}$ or a $\text{mirLoopControlVar}$. The number of iterations related to $\text{mirBasicIndVar}$ is the iteration counter of the main loop variable. The rest of the information needed for the application of loop reversal is contained in the structure of the variable, $\text{mirBasicIndVar}$, also presented in Listing 2.12.

The loop requirements have been defined based on the previous analysis of the possible loop variables contained within a loop and the related information available. Thus, the loop requirements that the engine needs to check in order to apply loop reversal, can be summarized as follows:

- The first restriction is that a loop must have a loop marker associated in order to be candidate for being reversed.
- There must be only one main loop variable: that can be either a $\text{mirIterCounter}$ or a $\text{mirLoopControlVar}$ and zero or more $\text{BasicIndVar}$. 
• The number of increment statements should be one, in all the considered cases of loop variables.

• In case a `mirLoopControlVar` is the main loop variable, the number of iterations needs to be computed. Therefore, all the variables needed for the calculation of the `IterCount` should be known and invariant.

Beside the checks on the loop variables that select the loops that the engine is able to reverse, some additional checks are done:

• Check if the `Init`, `Test` and `Incr` basic blocks are reconstructable, which means they do not have side effects.

• Loops containing a `Guard` basic block (see Listing 2.11) are not considered for the application of reversal.

• Neither the loops with more that one exit block are considered.

4.2.2 Program Flow

The `loopreversal` engine operates on each procedure of the compilation unit. It looks at the information of all the loops that have been identified by the `loopanalysis` engine, and annotated as a tree of loop markers. The engine tries to reverse one loop within each loop nest found. The program flow can be divided into four steps:

Get data dependence information: while traversing the tree of loop markers in preorder, the engine checks if there is any data dependence information attached the loops. If a data dependence graph is found in some loop, this implies that this loop is the outermost loop of an analyzable loop nest, thus a flag named `valid_nest` is set to true. Then, in this same step, the data dependence graph found is explored in order to check the validity of the loop nest, and the validity of each level to be reversed. The details of this checking are further explained.

Figure 4.2 illustrates the next three steps, which are performed while traversing the tree of loop markers in post-order. If the loop nest is still valid, then the legality and the applicability of loop reversal is checked. Only if these checks are passed, loop reversal is applied. Note that doing the check from bottom-up (post-order) gives priority to the innermost loops to be the one reversed within the loop nest.

Check legality of the loop: each loop belongs to a level within the loop nest, to determine if the specific loop can be reversed, it is necessary simply to check that the loop carries no dependences. During the process of exploring the data dependence graph, all the levels that are carrying dependences within the loop nest have been previously stored in a list named `forbidden_levels list`. Therefore, this step of the legality test consists of ensuring that the level of the loop that is being analyzed as candidate for reversal is not contained in the `forbidden_levels list`. If the loop is legal, then the applicability is checked, otherwise the next upper level is analyzed.

Check applicability of the transformation: the check of applicability consists of identifying within the candidate loop all the elements involved in the application of the loop reversal:
4.2 Implementation Details: Loop Reversal

Figure 4.2. Checks for each loop when traversing the loop marker tree in post-order
initial value, the step size, direction and number of iterations. If any of these elements are not known, or if the new expressions that are going to replace these values can not be generated, then it cannot be guaranteed that the application will be successful. Therefore, the loop cannot be reversed and the next loop is analyzed.

**Apply loop reversal**: this step consists of the traversal of the Init, Test and Incr basic blocks annotated in the loop marker (see listing 2.11). Within these basic blocks, the old expressions identified in the previous step are actually replaced by the new expressions also generated in the previous step.

### 4.2.3 Exploration of the Data Dependence Graph

When the tree of loop markers is traversed in preorder, the engine checks if there is any data dependence information attached to the loops. This information is contained in a data dependence graph (DDG), that needs to be analyzed to determine the validity of the loop nest and the validity of each level to be reversed. The DDG is inspected as depicted in Figure 4.3.

First the flag **lcdValid** (see Section 2.6.2) contained in the loop marker, is checked to determine if the data dependence information has been attached to the loop, if founded this make the loop nest valid.

In the data dependence graph, the list of edges is traversed. For the information annotated in each edge (see **mirLcdEdge** structure in listing 2.13), the function **checkValidity** checks the information referring to the levels where the dependence is carried. The actions taken in each

![Figure 4.3. Check for data dependence information flow diagram](image-url)
4.2 Implementation Details: Loop Reversal

case are summarized in 4.2, also it is shown the relation of the checked fields of the edges with the direction matrix.

Table 4.2: Implications for level information annotated in the mirlcdEdge structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Direction Matrix</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>(=,=,=)</td>
<td>valid_nest remains true</td>
</tr>
<tr>
<td>AnyLevel</td>
<td>(*)</td>
<td>valid_nest = false</td>
</tr>
<tr>
<td>Level</td>
<td>(&lt;,=,=)</td>
<td>add level to forbidden_levels list</td>
</tr>
</tbody>
</table>

The process of checking each edge of the DDG is equivalent to analyzing each row of the dependence direction matrix of the loop nest. Each row represents a dependence just as each edge represents a dependence. When a dependence is independent from the loops (the edge operator is Independent), then to reverse any of them does not violate the dependence, thus the loop nest remains valid. Any Level means that it is uncertain which of the loops is carrying the dependence, thus, this dependence invalidates the complete loop nest. Finally Level refers to a loop-carried dependence where the level is only one and known. The dependence is carried by the specified level. The level cannot be reversed and therefore this information is stored in the forbidden levels list. Note that at the end of this process, the status flags valid\_nest is set and a list of forbidden levels has been generated.

4.2.4 Additional Considerations

The mirlLoopMarker is extended with the following boolean variables in order to keep track of the process:

Listing 4.5: Extension of mirlLoopMarker structure

```
domain mirlLoopMarker [ref] :
<
    LegalReversal [init("FALSE")]: BOOL,
    ApplicableReversal [init("FALSE")]: BOOL,
    Reversed [init("FALSE")]: BOOL
>;
```

The CCMIR is also extended with a structure to handle the information of the induction variables. The need of these structures is due its dependence of the induction variables with main loop variable. To know the last value that the induction variable get when iterating the main loop, it is necessary to know the number of iteration. Only then the NewInit value of the induction variables can be generated. Therefore, the induction variables and the information needed are identified and stored in a list of structures called lrIndVar. Later, once all loop variables has been analyzed then the new expression for the induction variables are generated.

The new structure looks as follows:
4.3 Integration into a CoSy Compiler

For the integration of the new engine with the tool the following engine was added into a EDL file of the CoSy compiler.

Listing 4.7. Extension of the EDL file in the C-to-C compiler

```
engine class loopreversal(in u: mirUnit; in t: TarDes)
{
    engine skip: test;
    engine demote2: demote;
    region p(): mirProcGlobal;

    pipeline
    if ! skip()
        procana(u, t)
        // prepare loops
        loopprep(u, t)
    unit2proc(u, p())
    if mlfilter(p, u, t)
        pragmaswitch(p)
        lddepana(p, u, t)
        loopreversal(p, u, t)
}
```

This engine class describes the interaction and the dependence of the new engine `loopreversal` with other engines. First of all, the engine is receiving the complete compilation unit (mirUnit), but it works on a procedure level. Note the unit2proc(u, p) line in Listing 4.7, where each procedure is identified, and therefore the loop reversal engine operates over each of them.

Furthermore, in the implementation of the engine different levels of verbosity was added for debugging and testing, and also support with pragmas in order to force or to forbid the application of loop reversal for analysis purposes only.
Additionally, the loopreversal needs the presence specially of procana, loopprep and lcdepana engines in order to operate correctly.

4.4 Tests Setup

Once the loopreversal was integrated within the C-to-C compiler, the applicability of the transformation and a verification of the engine was done using the following applications from the MIBech and DSPStone benchmark suite: complex_multiply, complex_update, dot_product, convolution, real_update, mat1x3, n_real_updates, n_complex_updates, startup, fir, fir2dim, biquad_one_section, lms, fft, des, dijkstra, blowfish, adpcm, crc, jpeg, basicmath, bitcnts, matrix, biquad_N_sections, epic, h264_decoder, sieve, stringsearch, rijndael, sha. Additionally, the synthetic application (lrt_test) code used in the previous experiments was added to the set in order to verify the same results are obtained when applying the transformation automatically by means of the C-to-C compiler.

The three same processors ARM926EJS, Texas Instruments C6455, and Tensilica Diamond570 are used in these experiments, from each application in the benchmark, each target compiler can receive three different inputs:

- the original source code (.c)
- the output from the C-to-C compiler with loop reversal is enable
- the output of the C-to-C compiler where loop reversal is disable

In principle when loop reversal is the only transformation in the C-to-C compiler the first and the third item should result in the same performance. However, in the last case the original code is lower to CCMIR and bring up by means of the mirtoc engine. Since here the interest is to measure the effect only of the loopreversal engine and no its combination with the mirtoc engine, the numbers of speedup presented in the following sections are the comparison between the source code where loopreversal and mirtoc is transforming the code and the transformed code where only mirtoc was used, which means that in both cases the input to the target compiler is the output of the C-to-C compiler.

4.4.1 Results

The application of the loop reversal transformation is evaluated in all applications in the benchmark suite. From the total of 31 applications the statistic presented in Table 4.3 were obtained:

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total applications</td>
<td>31</td>
</tr>
<tr>
<td>Applications with Data Dependence Graphs</td>
<td>9</td>
</tr>
<tr>
<td>Applications where reversal was applied</td>
<td>9</td>
</tr>
</tbody>
</table>

From the total of applications 807 loops where analyzed and only 61 had a data dependence information available. For twenty-two applications no data dependence graphs were attached
to the loop structures, thus loop reversal can not be applied at all. Therefore, the set where loop reversal can be applied was reduced to nine applications, in all of them loops to apply the transformation were found. For the applications where data dependence graphs were found, some statistics were collected and presented in Table 4.4.

Table 4.4. Statistics on applications where data dependence graphs were found

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Loop Markers</th>
<th>DDG Found</th>
<th>Reversed Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>biquad_N_sections</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>epic</td>
<td>143</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>h264_decoder</td>
<td>377</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>sieve</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>stringsearch</td>
<td>28</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>rijndael</td>
<td>15</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>sha</td>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>lrt_test (reversal.c)</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>597</td>
<td>61</td>
<td>41</td>
</tr>
</tbody>
</table>

From all the applications 597 loop markers contained in these nine applications, only 61 had a valid data dependence graph. From those 61, for 41 loops the application of loop reversal was possible. Then, the reversed code and the only-micro transformed code were cross-compiled for every target, and executed in the corresponding simulator. On Table 4.5, the results of speedup for these 9 applications on each architecture are shown.

Table 4.5. Speedup obtained where loop reversal was applied

<table>
<thead>
<tr>
<th>Application</th>
<th>ARM926EJS</th>
<th>TIC6x</th>
<th>Diamond570</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>0,99893</td>
<td>1,00065</td>
<td>0,99938</td>
</tr>
<tr>
<td>biquad_N_sections</td>
<td>1,01149</td>
<td>0,99993</td>
<td>1,00000</td>
</tr>
<tr>
<td>epic</td>
<td>1,00427</td>
<td>1,00000</td>
<td>1,00064</td>
</tr>
<tr>
<td>h264_decoder</td>
<td>0,99702</td>
<td>1,00294</td>
<td>1,00005</td>
</tr>
<tr>
<td>sieve</td>
<td>1,00000</td>
<td>0,86020</td>
<td>0,99986</td>
</tr>
<tr>
<td>stringsearch</td>
<td>1,00028</td>
<td>1,00000</td>
<td>1,00000</td>
</tr>
<tr>
<td>rijndael</td>
<td>0,99932</td>
<td>0,99940</td>
<td>1,00191</td>
</tr>
<tr>
<td>sha</td>
<td>1,00000</td>
<td>1,00000</td>
<td>1,00000</td>
</tr>
<tr>
<td>lrt_test</td>
<td>1,02659</td>
<td>0,99995</td>
<td>3,96678</td>
</tr>
</tbody>
</table>

4.5 Summary

From the results obtained, it is shown that the opportunities to apply loop reversal has been reduced due to the lack of the data dependence information that is given by the standard CoSy engine ldcsepana. From the loops left 67% where possible to reversed. Additionally, loop
reversal is an enabling transformation, therefore non great improvements in the performance are expected. However, from the previous evaluation, it was concluded that the compiler for the Diamond570 can recognize opportunities where reversing a loop can enable a transformation such as permutation. However, according to the results shown in Table 4.5 for none of the loops reversed, another transformations was enabled. Except for the synthetic code which was also automatically reversed.
Chapter 5

Conclusions and Outlook

In this thesis an evaluation of the loop reordering transformations as source to source transformations was done in order to analyze their profitability and their applicability for modern embedded processors. They were identified as two different kind of transformations, the enhancing-locality transformation and the enabling transformations. Each transformation was manually executed on synthetic code and their performance was measured in three different targets. The analysis discusses the different effects of the transformations. The loop reordering transformations not only acts over the memory system, the resultant scheduling generated by the compiler and the exposure of parallelism varies when these transformations are applied.

The locality-enhancing transformations, loop permutation and tiling, gave improvements due to its effect on the locality, and a characterization of these transformations show that different program context factors and architectural factors determine the profitability. Results of loop permutation show the expected speedups in the case of the ARM and Texas Instruments processors. On the Tensilica Diamond 570T none speedup has seen, which leads to the conclusion that the target compiler has integrated the transformations. Tiling gave extra benefits over the profit obtained by loop permutation. It was also shown that the application of enabling transformations may constitute a great benefit, like in the case of the compiler of the Diamond 570T, where applying loop reversal alone yields a speedup of more than 8.

Encouraged by these results, a case study was performed to test the applicability of automated loop reordering transformations. For this loop reversal was used, because it yielded high speed up on at least one target platform. The transformation was implemented in the CoSy framework. When testing the new engine with the benchmarks it was shown that the applicability of loop reversal was reduced due to the number of non-analizable loops, for which the standard engine lcdepana can not make any annotation regarding the data dependence information.

In future work, the addition of more transformations like loop permutation, or the combination with loop reversal can be tested. From the analysis of the evaluation of the loop reordering transformation, it was seen that not only improvements but also degradation in the performance are obtained. Therefore, the application of these transformations must be selectively applied. Finally, a more in-depth study can be made regarding the understanding of the conflict misses behavior, that shows to be determinant for this kind of transformations.
Bibliography


